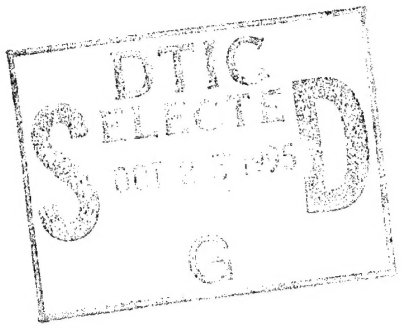
Exploiting Bandwidth to Reduce Average Memory Access Time in Scalable Multiprocessors

Ricardo Bianchini



Technical Report 582
April 1995

UNIVERSITY OF
ROCHESTER
COMPUTER SCIENCE

Exploiting Bandwidth to Reduce Average Memory Access Time in Scalable Multiprocessors

by

Ricardo Bianchini

Submitted in Partial Fulfillment

of the

Requirements for the Degree

Doctor of Philosophy

Supervised by

Professor Thomas J. LeBlanc

Department of Computer Science
The College
Arts and Sciences

University of Rochester
Rochester, New York

1995

This dissertation is dedicated to my father,
Evaristo O. R. Bianchini.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	April 1995	technical report

4. TITLE AND SUBTITLE

Exploiting Bandwidth to Reduce Average Memory Access Time in Scalable Multiprocessors

5. FUNDING NUMBERS

N00014-92-J-1801 / ARPA 8930

6. AUTHOR(S)

Ricardo Bianchini

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESSES

Computer Science Dept.
734 Computer Studies Bldg.
University of Rochester
Rochester NY 14627-0226

8. PERFORMING ORGANIZATION

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESSES(ES)

Office of Naval Research
Information Systems
Arlington VA 22217

ARPA
3701 N. Fairfax Drive
Arlington VA 22203

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

TR 582

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT

Distribution of this document is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

(see title page)

14. SUBJECT TERMS

communication bandwidth; communication latency; scalable multiprocessors; network and memory contention; latency tolerance

15. NUMBER OF PAGES

141 pages

16. PRICE CODE

free to sponsors; else $4.50

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
unclassified	unclassified	unclassified	UL

Curriculum Vitae

Ricardo Bianchini was born July 13, 1966, in Rio de Janeiro, Brazil. He attended the Federal University of Rio de Janeiro from 1984 to 1990 where he received a B.S. and an M.Sc. in Computer Science.

He entered the Ph.D. program at the University of Rochester in 1990 and received an M.Sc. in Computer Science in 1992. At Rochester, he has worked primarily with Prof. Thomas J. LeBlanc on topics related to high-performance parallel computing.

Acknowledgments

I would like to thank my friend and advisor, Tom LeBlanc, for having taught me almost everything I know about research and for having contributed heavily to the work described in this dissertation. Tom and his family have contributed significantly to make "the Rochester experience" a great one.

The other members of my thesis committee, Chris Brown, Charles Merriam, and Michael Scott, also deserve many thanks for the helpful discussions and suggestions that improved this dissertation. Chris and Michael contributed directly to my development as a researcher by working with me in several opportunities.

This department's staff (Tim Becker, Liud Bukys, Jill Forster, Ray Frank, Peggy Frantz, Pat Marshall, Peggy Meeker, Brad Miller, and Jim Roche) are the most competent I have ever seen and, along with the faculty, provide the best possible environment for research.

Perhaps no single experience contributed more to my graduate education than my five-month stay at MIT. I would like to thank Anant Agarwal for the opportunity to work with his group on the evaluation of the MIT Alewife multiprocessor. Many thanks also to all the members of Anant's group for having made my stay in Cambridge extremely enjoyable.

Besides being my good friends, Mark Crovella, Fredrik Dahlgren, Leonidas Kontothanassis, Jack Veenstra, and Bob Wisniewski, have had a significant influence on my work. Mark, Leonidas, and Jack have contributed to some of the work described in this dissertation. Leonidas deserves special mention as we have written several papers together. I like to think that we form a great team; at least, he always knows the stuff I don't. :-)

A few other fellow students helped me get through my first year in the PhD program (which coincided with my first year of marriage and my first year in the US). They are Karen Bogucz, Eliseu Chaves Filho, Bill Garrett, and Jonas Karlsson. Thanks for all the support and for being around when I needed to commiserate; I guess I had too much on my mind at the time. :-)

I would also like to thank my closest Brazilian (that's not quite right, let's say Portuguese-speaking) friends in Rochester, Mauricio Marengoni, Wagner Meira Jr., José Luiz Neves, and their families. They were part of my day-to-day life and were always around when I needed "to get out of the lab". Claudio Amorim, Eliseu Chaves Filho, Luiz Adauto Pessoa, and Cesar Quiroz also deserve many thanks for being in contact

with me when I felt like I was going to work myself to death, and for having helped me get back on track.

My family has been immensely supportive and encouraging throughout the years. My mother and step-father, Sandra and Mario Sampaio Ferraz, particularly, have been a constant source of encouragement and have made sure my vacations were always filled with love and fun.

Finally, I would like to thank my wife Marcia for never failing to take care of me. Marcia's love, encouragement, and patience are the reasons why I was able to complete this dissertation and get a PhD degree. However, the most important reason for my gratitude towards Marcia has nothing to do with my work; Marcia and our soon-to-be-born son have finally made me realize what's important in life, and it ain't computer science. :-)

The work reported in this dissertation was supported by Brazilian NUTES/UFRJ and CAPES/MEC (grant number 2038/90-2) fellowships, and by the Office of Naval Research Contract No. N00014-92-J-1801 (in conjunction with ARPA Research in Information Science and Technology - High Performance Computing, Software Science and Technology program, ARPA Order No. 8930).

Abstract

The overhead of remote memory accesses is a major impediment to achieving good application performance on scalable shared-memory multiprocessors. This dissertation explores ways in which to exploit network and memory bandwidth in order to reduce the average cost of memory accesses. We consider scenarios (1) where the remote access cost is dominated by contention, and (2) where the hardware provides abundant bandwidth and the remote access time is dominated by the unsaturated request/access/reply sequence of operations. We introduce and evaluate two techniques for increasing the effective bandwidth available to processors, software interleaving and eager combining. We also evaluate strategies for hiding the high cost of remote accesses, including several forms of prefetching and update-based coherence protocols. We use both analytic models and detailed simulations of multiprocessor systems to quantify the effectiveness of these techniques, and to provide insight into the potential and limitations of exploiting bandwidth to reduce average memory access cost.

Table of Contents

List of Figures

List of Tables

1 Introduction

Scalable shared-memory multiprocessors use hardware caches to reduce the average cost of a data access by storing data close to processors that need them. However, due to several reasons, not all memory references hit in the processor caches. A cache miss may require a remote memory access, which invariably takes a significant number of processor cycles to satisfy. This overhead is a major impediment to achieving good application performance on scalable multiprocessors.

Consider the simplified view of a scalable multiprocessor presented in figure 1.1. In this organization, which is similar to the Stanford DASH [Lenoski *et al.*, 1993] multiprocessor, satisfying a write miss remotely might not introduce any running time overhead. Write buffers and relaxed consistency models [Dubois *et al.*, 1988] allow for write requests to be performed without stalling the processor and, therefore, hide most of the cost of remote writes. The overhead of read misses satisfied remotely cannot be hidden however, since read misses always stall the processor. This stall period can significantly affect performance, since satisfying a read miss remotely requires (at least) a local bus access, a trip through the network to the node that is to provide the data block, a bus access on that node, a memory access to fetch the data, a trip back across the network, and another local bus access to fill the cache line.

On current multiprocessors, a read miss can take from about 50 processor cycles on the MIT Alewife machine [Agarwal *et al.*, 1995] to about 250 processor cycles on the Kendall Square Research KSR1 multiprocessor [Kendall Square Research Corp., 1992]. Given that RISC microprocessors can execute on average close to one instruction per processor cycle and a cache hit usually takes a single cycle, a processor taking a read miss could execute tens or hundreds of instructions during the time the miss is outstanding. This scenario becomes even worse if we consider the latest developments in superscalar microprocessors [Smith and Weiss, 1994]. As processor speeds continue to improve at a dramatic rate, the relative importance of remote accesses will continue to grow.

The question then is how to reduce the average memory access overhead. Many researchers have been investigating techniques intended to reduce this form of overhead. These techniques can be divided into three groups: techniques that reduce the number of remote accesses, techniques that reduce the cost of individual remote accesses, and techniques that overlap remote accesses and computation and, thereby, hide (or tolerate) the overhead of remote accesses. Our work considers techniques belonging to the two

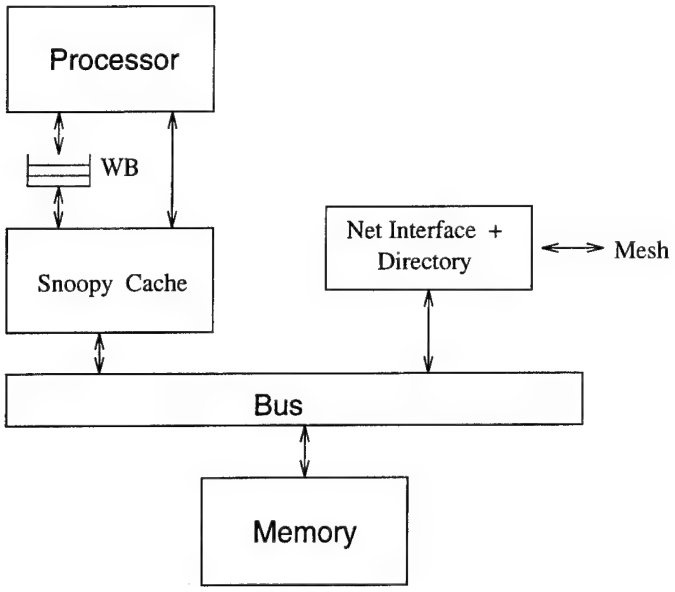


Figure 1.1: Example of multiprocessor architecture.

latter categories. We consider two different scenarios: (1) where the remote access time is dominated by memory or network contention; and (2) where the hardware provides high remote access bandwidth and the remote access time is dominated by the duration of the remote (unsaturated) request/access/reply sequence of operations.

This dissertation reports on the performance of novel techniques that increase the effective bandwidth available to processors so as to reduce the cost of remote accesses during periods of contention, and on the effectiveness of aggressively exploiting high-bandwidth resources to hide the overhead of remote accesses. We use both analytic models and detailed simulations of scalable cache-coherent multiprocessor systems to evaluate these techniques, and to provide insight into the potential and limitations of using excess network and memory bandwidth as a solution to the remote access overhead problem.

1.1 Related Work

There has been a significant amount of research on strategies for reducing the average memory access time in scalable multiprocessors. In this section we summarize this research and relate it to the work presented in this dissertation. Each of the following chapters contains a more specific comparison between the work presented in the chapter and other related approaches.

1.1.1 Reducing the Number of Remote Accesses

Hardware caches provide the most effective way to reduce the number of remote accesses in a multiprocessor [Gupta *et al.*, 1991]. However, caches are not always successful at

keeping the number of remote accesses down to an acceptable level. Thus, several techniques have been proposed to improve the performance of hardware caches. Chief among these is blocking (or tiling) of data accesses [Lam *et al.*, 1991], whereby successive references to the same data are grouped temporally close to each other in parallel loops.

Cache-conscious thread and data allocation techniques are also very effective strategies for improving locality of reference. Among these strategies, loop transformations (such as loop interchange [Padua and Wolfe, 1986]) are used to improve the spatial locality of multi-word cache line accesses, while false-sharing-elimination techniques (such as indirection [Eggers and Jeremiassen, 1991] and software caching [Bianchini and LeBlanc, 1992]) are used to improve processor locality. Affinity scheduling [Vaswani and Zahorjan, 1991; Markatos and LeBlanc, 1994] is used to co-locate threads and the data they use.

Implementations of caching in software can also reduce the number of remote accesses significantly [Bolosky *et al.*, 1989; Cox and Fowler, 1989; LaRowe Jr. and Ellis, 1991], by creating copies of read-only pages in the processors' local memories and moving pages between these local memories on writes.

1.1.2 Reducing the Cost of Individual Remote Accesses

Several techniques have been proposed for reducing the overhead of individual remote accesses. In interconnection network design, for instance, wormhole routing [Dally, 1990] and low-degree topologies [Dally, 1990; Agarwal, 1991] are currently used commercially, as these approaches have been shown to reduce the overhead of network transfers. Adaptive routing techniques [Ni and McKinley, 1993] have been proposed to alleviate network congestion problems.

A large body of work has also been done on memory-contention-alleviating techniques. The majority of these techniques assume special hardware support, such as interleaved memory and multi-stage interconnection networks (MINs) with combining of memory references. In scalar multiprocessors, memory interleaving is sometimes used to alleviate contention caused by many processors trying to access the same memory module. The IBM RP3 [Pfister *et al.*, 1985], the Tera computer [Alverson *et al.*, 1990], and the BBN TC2000 [BBN, 1989] are examples of machines that use interleaving of memory addresses.

While memory interleaving tackles memory contention caused by processors trying to access data in nearby addresses, combining multi-stage interconnection networks try to avoid contention due to accesses to the same datum. The idea is to combine references to the hot datum as requests flow through the interconnection network. A few multiprocessors have implemented combining switches, including the NYU Ultracomputer [Gottlieb *et al.*, 1983] and the IBM RP3 [Pfister *et al.*, 1985].

Among the contention-alleviating techniques that do not require special hardware, software broadcasting stands out as the most frequently used technique. Software broadcasting typically employs a logarithmic distribution of data, where the source processor sends the data to N other processors, each of which pass the data on to N other processors, until all processors receive the data. Two other related techniques for alleviating

memory contention are software combining trees [Yew *et al.*, 1987] and data replication. Software combining trees are analogous to hardware combining networks, and incorporate logarithmic broadcasting. As in combining networks, requests for shared data flow from the leaves of the tree to the root, and the data flows from the root down to the leaves. As in logarithmic broadcasting, the data flows down the tree in $O(\log P)$ steps (where P is the number of processors), and the width of the tree limits the potential for memory contention.

We can also limit memory (and possibly network) contention by replicating data across multiple memory modules. By distributing the requests for data evenly among the copies, we can reduce or eliminate memory contention for the original copy.

In this dissertation, we propose and evaluate novel techniques that use data distribution (*Software Interleaving*), and replication and combining (*Eager Combining*) to increase the effective bandwidth available to processor requests, thereby allowing for reductions in the cost of remote accesses in the presence of contention.

1.1.3 Overlapping Remote Accesses and Computation

The most important techniques for overlapping remote accesses and computation are relaxed consistency, update-based coherence protocols, prefetching, and multiple-context processors. In these techniques, remote accesses happen in the "background", i.e. the processor does not stall waiting for the accesses to complete.

Relaxed consistency models [Dubois *et al.*, 1988; Gharachorloo *et al.*, 1990] reduce the cost of remote writes, as writes by a processor are allowed to be seen out of order by other processors. Thus, relaxed consistency protocols allow a writing processor to continue its normal execution before receiving all the write acknowledgements from other processors. Only synchronization operations must be seen in the same order by all processors under a relaxed consistency protocol.

In the presence of write buffers and relaxed consistency, update-based protocols [Archibald and Baer, 1986; Lilja, 1993] reduce the average cost of reads by eagerly multicasting a write to all processors sharing the written cache block.

Multiple-context processors [Smith, 1978b; Alverson *et al.*, 1990; Agarwal, 1992] tolerate the overhead of remote reads and writes by switching between threads of execution when remote memory accesses must occur. Thus, these processors can execute useful work while remote accesses are pending.

Prefetching [Callahan *et al.*, 1991; Mowry and Gupta, 1991; Dahlgren *et al.*, 1993] also tolerates the overhead of remote read and write accesses, by issuing these requests well before they are actually needed by the processor. Requests must be issued enough in advance so that they can complete before the processor has to stall.

The performance of these techniques is affected by the amount of remote access bandwidth available to processors, since the techniques rely on overlapping communication and computation to avoid processor stalls, and the performance of communication operations depends on bandwidth. In this dissertation we examine techniques that overlap remote accesses and computation to determine how aggressively they can be applied in the presence of abundant remote access bandwidth.

The Tera computer [Alverson *et al.*, 1990] also exploits high bandwidth to tolerate the overhead of remote accesses. When completed, the Tera multiprocessor will provide extremely high network and memory bandwidth, resulting from an aggressively-designed direct network, and memory interleaving and randomization. Overlapping of remote accesses will be implemented with multiple-context processors; each processor will switch between hardware contexts on every cycle. Instead of a cache, each processor will have 128 (the maximum number of contexts) different sets of registers. Even though our work does not focus on multiple-context processors, our experience with other techniques for overlapping communication and computation offers a cautionary note for multiprocessors like the Tera. Our work suggests that the performance of such machines depends critically on the ability to distribute memory references uniformly. Even a slight case of non-uniform references can degrade performance significantly, due to the enormous demands put on the memory sub-system by the large number of contexts and the lack of caches in the machine.

This dissertation also investigates whether excess bandwidth is enough to make techniques currently seen as suboptimal (partly as a result of insufficient bandwidth) outperform their competitors. More specifically, we consider various forms of hardware prefetching as alternatives to software prefetching, and a write-update protocol as an alternative to write-invalidate protocols.

1.2 Methodology

A fundamental part of our work is understanding the effect of architectural variations on the performance of real applications running on scalable cache-coherent multiprocessors with release consistency [Gharachorloo *et al.*, 1990]. Thus, our studies are based on execution-driven and trace-driven simulation. Each node in the simulated machine contains (at least) a single processor, a cache, local and directory memory, and a network interface. We simulate the processor at the level of individual instructions, the network at the level of individual flits (whenever necessary), and the memory modules at the level of individual requests. In all of our studies, we assume that remote accesses are 30-100 times more expensive than a cache hit. We present more details about the simulation infrastructure and application workload used for each of our studies in their corresponding chapters.

1.3 Overview of the Dissertation

The remainder of the dissertation is organized as follows. Chapters 2 and 3 describe two techniques that can increase effective memory and network bandwidth and, as a result, can alleviate the effect of contention on remote access overhead. In chapters 4 and 5, we investigate whether fetching large amounts of data on a cache miss can provide performance benefits under high network and memory bandwidth assumptions. Chapter 6 presents the results of our investigation of whether high memory and network bandwidth can justify the use of a write-update coherence protocol, which places an

enormous burden on the network and the memories in the form of update transactions. Chapter 7 presents our conclusions.

2 Reducing Contention with Eager Combining

One common cause of poor performance in large-scale shared-memory multiprocessors is limited memory or interconnection network bandwidth. Even well-designed machines can exhaust the available bandwidth when a program issues an excessive number of remote memory accesses or when remote accesses are distributed non-uniformly. While techniques for improving locality of reference are often successful at reducing the number of remote references, a non-uniform distribution of references may still result, which can cause contention both in the interconnection network and at remote memories.

A non-uniform distribution of remote accesses can be caused by a variety of common data sharing patterns. Centralized spin locks, for example, force all processors to access the memory module containing the lock data structure. In many linear algebra algorithms, including straightforward parallelizations of Gaussian elimination and LU decomposition, a single processor writes a row of a matrix which must then be read by every other processor. Classical graph algorithms, such as transitive closure and all-pairs shortest path, exhibit this same structure. Many optimization algorithms maintain the best global solution found so far in a global location, and require all processors to access that location each time a new solution is found. These sharing patterns can introduce enormous network and memory contention on large-scale machines, since a large number of processors may need to access a single memory module simultaneously.

These sharing patterns are all specific instances of producer/consumer data [Bennett *et al.*, 1990], where data is written by one processor and then read by many processors. Several techniques have been developed for reducing the contention caused by producer/consumer sharing. Hardware combining, as in the IBM RP3 [Pfister *et al.*, 1985], can alleviate contention for spin locks by combining requests to a single memory location. Alternatively, spin locks can be implemented so as to spin on local memory only, thereby eliminating most remote references associated with synchronization. Optimization algorithms can avoid contention by examining or updating the global solution infrequently. Linear algebra algorithms can exploit the properties of numerical equations to improve locality of reference, and as a side-effect eliminate most producer/consumer sharing [Gallivan *et al.*, 1990].

Although most of these techniques reduce contention and improve locality of reference, they may introduce significant complexity in the algorithm, and do not generalize to all producer/consumer sharing. For example, the block algorithms used in linear

algebra are quite complex (compared to the straightforward algorithms), and do not generalize to graph algorithms. Optimization algorithms that avoid using the global solution so as to improve locality of reference may adversely affect the search [Bianchini and Brown, 1993]. Given the frequency with which producer/consumer sharing arises, the performance implications for large-scale machines, and the complexity of eliminating this sharing pattern on a case-by-case basis, a general solution is desirable.

As a general solution to the problem of producer/consumer data sharing, we propose a new coherence protocol, called *eager combining*. The protocol is an extension of the DASH write-invalidate protocol [Lenoski *et al.*, 1990]. Our protocol incorporates ideas from software combining trees [Yew *et al.*, 1987] and eager sharing [Wittie and Maples, 1989] to distribute requests for producer/consumer data throughout the machine. The protocol replicates a producer's data among multiple memory modules, thereby effectively increasing both the memory and network bandwidth of the producer, and dramatically decreasing the remote access overhead experienced by consumers.

Munin [Bennett *et al.*, 1990], a runtime system for networks of workstations, also includes a coherence protocol for producer/consumer data. The Munin protocol is based on object replication and write update. Munin runs on a network of SUN workstations, and exploits the broadcast capabilities of the Ethernet. Our protocol is intended for use in large-scale shared-memory machines with no hardware broadcast mechanism.

Most previous studies of non-uniform addressing and contention have focused primarily on the effects of hot spots on multistage interconnection networks [Pfister and Norton, 1985; Patel and Harrison, 1988]. These studies focused on eliminating tree saturation, and used synthetic applications for experiments. Our work focuses on producer/consumer sharing in direct-connected, distributed-shared-memory machines (such as the Stanford DASH [Lenoski *et al.*, 1993] and MIT Alewife [Agarwal *et al.*, 1995]), and our experiments are based on real application programs.

In this chapter we examine the effect of producer/consumer data on network and memory contention. We use detailed execution-driven simulation of parallel programs to quantify the performance impact of producer/consumer data as a function of the network and memory bandwidth and the number of processors in the machine. Our simulation results show that eager combining can improve performance (and effective bandwidth) by a factor of 4 or more when used for programs with producer/consumer data.

The remainder of this chapter is organized as follows. In Section 2.1 we describe our multiprocessor simulator and application suite. In Section 2.2 we quantify the effects of contention as a function of network and memory bandwidth. Section 2.3 describes the protocol in detail. In Section 2.4 we evaluate the performance of our applications under the eager combining protocol, and compare it to software logarithmic broadcasting. We conclude, in Section 2.5, with a summary of our results.

Machine	Latency/Word	Q Size	Mem Bwidth
Cont-free	1/2/4 cycles	0	Infinite
High	1 cycle	16	400 MB/sec
Medium	2 cycles	16	200 MB/sec
Low	4 cycles	16	100 MB/sec

Table 2.1: Memory bandwidth levels used in simulated machine.

2.1 Methodology and Workload

We use an on-line, execution-driven simulator that exploits a mixture of interpretation and native execution to simulate unmodified MIPS R3000 object code. The simulator is divided into two parts, an event generator [Veenstra and Fowler, 1994a] and an event executor. The event generator simulates the processor and registers, and calls the event executor on every memory reference. The event executor processes each reference, returning immediately on a cache hit, generating a memory request in case of a cache miss to local memory, or passing the reference through the interconnection network to a remote node otherwise. The event executor determines which processors block awaiting remote references and which processors continue to execute. We simulate events at the level of processor cycles; all simulation parameters and results are expressed in terms of processor cycles. Each node in the simulated machine contains a single processor, cache memory, local memory, directory memory, and network interface. Each processor has an infinite, write-back cache with 32-byte blocks. Caches are kept coherent using our implementation of the DASH protocol [Lenoski *et al.*, 1990] with release consistency.

Throughout this chapter we refer to the ensemble of addressable local memory and directory memory at each node as a "memory module." We simulate three types of memory systems: memory modules that respond with a negative acknowledgement if the memory is busy; memory modules that queue requests (coming either from the cache or network interface) when the memory is busy; and infinitely-ported contention-free memory modules. An infinitely-ported memory module can satisfy an arbitrary number of memory requests simultaneously, but each request is delayed by the memory service time (latency per cache block). We vary the memory latency per block between 8 and 32 cycles. Each shared address space consists of 4KB pages. The pages of an address space are allocated to processors in round-robin fashion. The memory latency and bandwidth parameters used in our experiments are described in table 2.1.

The interconnection network is a bi-directional wormhole-routed mesh, with dimension-ordered routing. At any node of the machine, the connection between the switch node and the processing node's network interface has the same bandwidth as any network link. Thus, we will refer to the bandwidth of that connection as the *network bandwidth per node*. Switch nodes introduce a 2-cycle delay to the header of each message. In our finite-bandwidth networks (derived from the Alewife cycle-by-cycle network simulator), contention for network links and buffers is fully captured. The network interface has a queue for out-going messages (triggered either by the cache or the

Machine	Path Width	Bi-dir Link Bwidth
Cont-free	32/16/8 bits	Infinite
High	32 bits	800 MB/sec
Medium	16 bits	400 MB/sec
Low	8 bits	200 MB/sec

Table 2.2: Network bandwidth levels used in simulated machine.

memory module in the node). For comparison purposes, we also implement an idealized, contention-free network, where the link width determines the bandwidth available to an individual packet, but any number of packets can traverse the same link or be stored in the same buffer simultaneously. The network bandwidth parameters used in our experiments are described in table 2.2.

Our application workload consists of five programs with producer/consumer data sharing: two linear algebra applications (Gaussian elimination and matrix inversion), two graph algorithms (transitive closure and all-pairs-shortest-paths), and an optimization algorithm (simulated annealing).

Our linear algebra applications are similar in that, during each phase of the computation, processors need access to a pivot row of the matrix describing the linear equations. This pivot row is written by one processor and then read by every other processor. In our graph applications, processors also require access to a pivot row of the adjacency matrix representing the graph. The pivot row is written by one processor and then read by many other (possibly all) processors. In all the applications, the elements of a matrix row are allocated to consecutive addresses in a single memory module, so all processors direct a request to the same memory module after synchronizing. In our simulated annealing program, processors search the solution space independently, periodically examining the current best-known solution, which is stored in a global location.

Our implementation of Gaussian elimination is similar in structure to the LU-decomposition application in [Mowry and Gupta, 1991]. The input is a 512×512 matrix of linear equations; we use locks for synchronization. Our implementation of matrix inversion also uses an input matrix of size 512×512, but synchronizes with barriers. The input graph for transitive closure has 768 vertices, and each vertex is connected to each other vertex with probability 0.5; we use locks for synchronization. We use a straightforward parallelization of Warshall-Floyd's algorithm to compute the all-pairs shortest paths of a graph; the input graph has 512 vertices, and each vertex is connected to each other vertex with probability 0.5. We use barrier synchronization for all-pairs. Our simulated annealing program finds the minimum of a complex function of 8 variables.

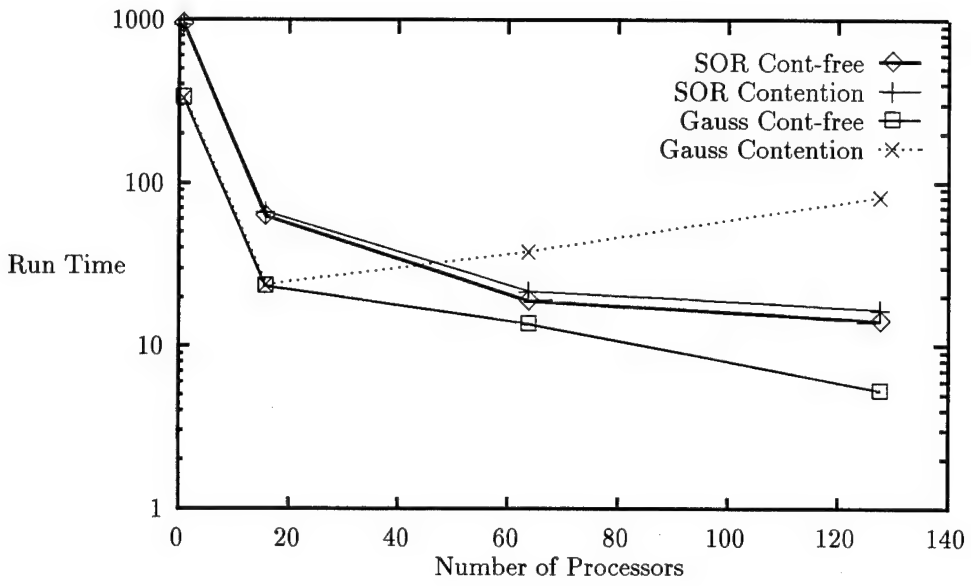


Figure 2.1: Running time (M cycles) of SOR and Gaussian elimination.

2.2 Effect of Bandwidth on Contention

In this section we quantify the effects of contention that result from producer/consumer sharing as a function of network and memory bandwidth.

2.2.1 Effect of Non-Uniform Distribution of References

Figure 2.1 compares the running time of SOR and Gaussian elimination on a multiprocessor with low memory and network bandwidth (200 MB/second, assuming a 100 MHz clock) to the running time on an ideal contention-free multiprocessor. SOR implements successive over-relaxation of a 768×768 matrix. Blocks of rows of the matrix are assigned to each processor. Processors work roughly independently of each other; communication only takes place when processors update the elements of their boundary rows. The application is executed for 50 iterations, which are divided into pairs of phases separated by barriers.

From the figure, we observe that the running time of SOR is not affected by contention; the machine with limited memory and network bandwidth performs comparably to the contention-free machine. The running time of Gaussian elimination is affected by contention however; the program can utilize fewer than 20 processors on the limited-bandwidth machine, but can exploit over 120 processors on the contention-free machine.

Both SOR and Gaussian elimination have very low miss rates – around 2% on 128 processors. Thus, both programs have good locality of reference. In general, we would expect programs with very low miss rates to perform well on large-scale machines

Network	Memory Bwidth											
Bwidth	High				Medium				Low			
	rt	cfrt	rl	cfrl	rt	cfrt	rl	cfrl	rt	cfrt	rl	cfrl
High	22.4	22.3	55.3	50.2	22.5	22.4	62.9	56.1	23.1	22.7	89.6	68.0
Medium	22.7	22.5	70.8	59.4	22.8	22.6	76.1	65.3	23.3	22.9	99.4	77.0
Low	23.7	22.9	119.3	80.2	23.7	23.0	125.0	85.9	23.9	23.3	137.3	97.7

Table 2.3: Running times and avg. latencies of Gauss (with and without contention) on 16 processors.

Network	Memory Bwidth											
Bwidth	High				Medium				Low			
	rt	cfrt	rl	cfrl	rt	cfrt	rl	cfrl	rt	cfrt	rl	cfrl
High	23.8	4.8	1201	81.3	29.4	4.9	1507	88.9	54.1	5.2	2847	103.9
Medium	43.2	5.0	2366	91.7	43.4	5.1	2368	99.1	60.4	5.4	3233	114.1
Low	82.4	5.3	4656	114.9	81.3	5.5	4651	122.3	82.8	5.7	4676	137.4

Table 2.4: Running times and avg. latencies of Gauss (with and without contention) on 128 processors.

(assuming sufficient parallelism and appropriate synchronization). The problem with Gaussian elimination is not the number of remote references, it is the non-uniform distribution of those references. Programs with good locality and nearest-neighbor communication, like SOR, do not saturate the network or memories. Programs with good locality and a highly non-uniform distribution of references can saturate both the network and memories.

This experiment shows that it may be difficult to design a multiprocessor that can run all classes of applications efficiently. Even well-designed machines, with seemingly sufficient network and memory bandwidth, may perform poorly for a large class of applications due to bandwidth limitations.

2.2.2 Scaling the Number of Processors

In tables 2.3-2.4 we examine the aggregate effect of the two types of contention on the performance of Gaussian elimination using realistic scenarios of limited network and memory bandwidth. These tables present the running time of the program on machines with finite network and memory bandwidth (rt), the running time in the absence of network and memory contention (cfrt), the average remote access latency on the realistic machines (rl), and the average remote access latency on the contention-free machines (cfrl). Running times are given in millions of cycles.

It is interesting to observe the evolution of the contention problem as we increase the

size of the machine from 16 to 64 to 128 processors. On 16 processors, contention for the network and memories is virtually nonexistent. In each case the running time under limited bandwidth is roughly the same as the running time on a contention-free machine. In addition, the difference in running time between the machine with high memory and network bandwidth and the machine with low memory and network bandwidth is only about 7%. Although the average remote access latency on these machines differs by a factor of 2-3, the miss rate is small enough that this factor does not significantly affect running time.

On 64 processors (not shown), the running time on the limited bandwidth machines is substantially higher than the running time on the corresponding contention-free machines (by 30% on the machine with the highest memory and network bandwidth and 470% on the machine with the lowest memory and network bandwidth). Whereas a factor of four increase in processors (from 16 to 64 processors) produces a factor of 2.5 improvement in running time on the machine with high memory and network bandwidth, the same increase in processors results in a higher running time on the machines with low memory or network bandwidth, and no significant increase in running time on the machines with medium network bandwidth. The effect of contention is illustrated most dramatically by the average remote access latency, which increases from 123 cycles to over 1800 cycles on the machine with low memory and network bandwidth.

These trends continue as we increase the number of processors from 64 to 128. In the absence of contention, a factor of two increase in processors produces a factor of 1.4 improvement in running time on most of the machines. When contention is included, the running time increases by a factor of 3 or more in nearly every case. Even on the high bandwidth machines, the remote access latency is well over 1200 cycles.

As we reduce memory bandwidth and keep the network bandwidth constant (that is, as we move across a row of the tables), running time remains roughly constant until the memory bandwidth becomes strictly less than the network bandwidth. At that point, the running time increases, since it is dominated by memory contention effects. If we increase the network bandwidth while keeping the memory bandwidth fixed (that is, as we move up a column in the tables), the running time improves greatly whenever memory bandwidth is greater than or equal to network bandwidth.

These results suggest that the usual notion of balanced architecture exhibits a serious bottleneck when faced with producer/consumer sharing. When memory and (unidirectional) network bandwidth are comparable, the producer's network link cannot keep up with the traffic, in part because the network link must transfer more data than the memory (including message headers, addresses, and operation codes), and also because the memory and cache controller compete for access to the network at each node of the multiprocessor.

These results show that producer/consumer data sharing can introduce enormous contention on large-scale machines. The problem is that only an inordinate amount of bandwidth can eliminate the contention levels observed here. Rather than build machines with sufficient bandwidth to handle extreme cases of non-uniform distributions of references, we consider ways to provide higher *effective* bandwidth.

2.3 Eager Combining Protocol

In this section we describe a coherence protocol that implements data replication for hot spots caused by widespread producer/consumer sharing. Our goal is to increase effective memory bandwidth and decrease the need for network bandwidth in direct-connected, distributed-shared-memory multiprocessors.

We assume certain physical address ranges are marked *hot*, and these addresses are treated specially by the coherence protocol. Our basic approach is to designate a fixed number of "server nodes" for each hot physical page, assigning to each server some subset of the remaining nodes as clients. The protocol uses eager sharing to distribute data to servers, which then satisfy requests from multiple client nodes. Multiple requests that cannot be satisfied immediately by a server are combined to reduce the traffic directed to a hot spot. The protocol assumes the release consistency model. Since our approach incorporates the properties of both eager sharing and combining trees, we call it *eager combining.*

We use the DASH cache coherence protocol [Lenoski *et al.*, 1990] as a starting point for our eager combining protocol. Each data block in DASH is assigned to a memory module, and that module's node is referred to as the data block's *home node*. In the eager combining version of the protocol, we designate a fixed number of *server nodes* for each hot data block, which are determined statically from the physical page number. As with regular data blocks, hot data blocks can be in one of three states: *uncached, read-shared,* and *modified.* We make three modifications to the DASH protocol in order to handle hot data blocks:

- Reads to a hot data block are directed to a server rather than to the home node;
- When a hot data block makes a transition from *modified* to *read-shared,* the block's home node multicasts the data to the block's servers;
- When a hot data block makes a transition from *read-shared* to *modified,* the clients and the servers must have their copies of the block invalidated.

The following sections describe each type of transaction in detail. Since our experiments assume infinite processor caches, the following description omits the details of how to handle cache replacements.

2.3.1 Read Requests

When a node makes a read request to a hot data block, the request goes directly to the proper server, which is selected based on the requester's node number and the physical page number. If the server has an unmodified copy of the block, it stores information to the effect that the requesting processor is a new client and sends the block to the requester. If the server does not have a copy of the block, or if the server has modified its copy of the block, the server marks the requester as a client and forwards the client's request to the home node. Subsequent requests from other clients for the same data block are queued at the server until it receives the block from the home node.

Upon receiving a forwarded read request, the home node proceeds according to the state of the data block. If the block is in the *read-shared* state, the home node sends the block to the client directly. If the block is in the *modified* state, the home node forwards the request to the current owner of the block. As in the DASH protocol, the owner transmits the data block to both the requester and the home node. On receiving the updated contents of the block, the home node sends a copy to each of the servers for the block, and sets the state of the block to *read-shared*. The multicast from the home node to the servers can be overlapped with computation on all nodes.

It is important to note that the home node does not multicast a data block to its servers each time the block is written. The multicast takes place only on the transition from *modified* to *read-shared*. Thus, we avoid eager sharing of partially modified data blocks. Nonetheless, eager combining could exacerbate any adverse performance effects caused by fine-grain sharing and false sharing.

A multicast and the corresponding transition to the *read-shared* state are also performed by the home node upon receiving a forwarded read request for an *uncached* hot data block.

2.3.2 Write Requests

When a client issues a write to a hot data block, a request for ownership (and in some cases, data) is sent directly to the home node, bypassing the server. On receiving this request, the home node proceeds according to the current state of the data block. If the block is in the *modified* state, the protocol proceeds almost exactly as in DASH: the request is forwarded to the current owner of the block, which transfers ownership (and perhaps data) to the requesting node, and requests that the home node update the ownership of the block. Although this latter request does not generate an acknowledgement in the DASH protocol, the eager combining protocol does require an acknowledgement from the home node to the previous owner of the block, so as to avoid sending ownership update messages through servers to the home node. [1]

If the write finds the data block in the *read-shared* state, the home node must invalidate all copies, both in the servers and clients. To implement these invalidations, the home node sends invalidation messages to servers, who then pass on invalidations to their clients. In this scheme, the directory information in the home node is consistent with respect to the state of a data block and the number of servers, but not the number of clients. When a write request reaches the home node, the home sends the data to the new owner, and tells the new owner the number of servers caching copies of the data block. The home node sends invalidation messages to each sever, which then send invalidation messages to each client. When the clients have all acknowledged the invalidation to their server, the server sends an acknowledgement to the new owner. The

[1] The reason for this extra acknowledgement message is that the ownership update message and a subsequent read request by the same processor may arrive out-of-order at the home node, due to the different paths these messages take through the network. This message does not have a noticeable impact on performance, since the extra acknowledgement is not in the critical path of the processors.

Consistency	Protocol	Tot Messages	Num Hops
Sequential	EC	(2S + 2(C - SC)) + 2	(2S + 2(C - SC)) + 2
	DASH-like	2C + 2	2C + 2
Release	EC	(2S + 2(C - SC)) + 2	2
	DASH	2C + 2	2

Table 2.5: Messages transferred in coherence actions (read-shared to modified).

home node invalidates the previous owner of the data block, which then acknowledges the new owner directly. [2]

2.3.3 Protocol Overhead

Eager combining is not without costs. It introduces additional messages and requires extra space for servers to keep track of their clients. We will now consider how much overhead is associated with the protocol.

Tables 2.5 and 2.6 present a comparison between the number of messages involved in the DASH protocol and in eager combining. In these tables, S stands for the number of servers per hot data block, C is the total number of clients of the hot block, and SC is the number of servers that are also clients of the hot block. The number of hops referred to in the tables is the number of messages in the critical path of the protocol. As observed in the tables, eager combining may employ more messages than the DASH protocol when making a transition from *read-shared* to *modified* or *modified* to *read-shared*, because hot data blocks are sent to servers whether or not the servers use the data, and both the servers and clients must be kept consistent. These extra messages are unlikely to be a serious problem however, since we expect hot data blocks to be accessed by most processors (including the servers), and the extra messages required to replicate data to servers can be overlapped under any consistency model. Also, it is not necessary to wait for invalidation acknowledgements if sequential consistency is not required. Therefore, under a relaxed consistency model, eager combining does not impose significantly greater communication latency than the DASH protocol. As shown by the number of hops in the critical path in tables 2.5 and 2.6, eager combining and DASH require roughly the same number of hops for each state transition, under the assumption that each server actually requires the data it provides to its clients.

It may seem that replicating cache lines in a large number of servers consumes an excessive amount of cache space, but this is not so. Assuming an even distribution of servers throughout the machine, the maximum additional cache space needed per processor is *HotDataSize * NumServers / NumProcs*, where *HotDataSize* is the size of the hot data, *NumServers* is the number of servers per hot cache block, and *NumProcs* is the number of processors in the machine. This is a worst-case analysis however; in

[2]The home node must invalidate the previous owner since the identity of this owner is unknown to the servers.

Protocol	Tot Messages	Num Hops
EC	S + 5	4
DASH	4	3

Table 2.6: Messages transferred in sharing actions (modified to read-shared).

practice, a server need only store the hot data currently being referenced by clients. In addition, under the assumption that servers are also clients for hot data, then each server needs a copy of the data anyway. Furthermore, the extra cache space devoted to copies of hot data in servers is a small percentage of the cache space devoted to caching application data. In short, the space overhead of servers is not an impediment to data replication as used in eager combining.

Eager combining also requires additional "directory" space for servers to keep track of clients. This extra space amounts to a vector of *NumProcs / NumServers* bits per cache block, in which each bit represents a client sharing the block. We believe that this overhead is justified by the performance advantages of the protocol, which will be demonstrated in the next section.

2.4 Performance Evaluation

In this section we evaluate the eager combining protocol. In our simulations of eager combining, requests are rejected (and must be reissued) when the producer's memory module is busy.

2.4.1 Comparison with DASH Protocol

In the experiments described in this section, we consider balanced architectures, where the memory and (uni-directional) network bandwidth are equivalent. Tables 2.7 and 2.8 present the running time of the applications (in millions of cycles) on machines with the DASH coherence protocol (rt), eager combining (ECrt), and the idealized, contention-free machine (cfrt) on 64 and 128 processors, respectively.

On the 64-processor machine we use 4 servers per cache block; we use 8 servers per cache block on the 128-processor machine. All the simulations assume release consistency.

Table 2.7 shows that eager combining significantly improves the performance of all the applications on the low-bandwidth machine with 64 processors. The running time of Gaussian elimination, all-pairs, and simulated annealing is improved by a factor of 2-3. Transitive closure and matrix inversion also exhibit improved performance, although the gains are not as substantial.

Substantial improvements are also possible on the medium-bandwidth machine for these applications. The improvements are not as great on the high-bandwidth machine,

Application	Bandwidth								
	High			Medium			Low		
	rt	ECrt	cfrt	rt	ECrt	cfrt	rt	ECrt	cfrt
`Gauss`	9.0	7.6	6.9	19.0	8.9	7.4	37.9	13.3	8.0
`Matinv`	65.8	62.4	58.1	76.1	67.4	59.7	99.9	80.5	63.4
`Tclosure`	47.7	43.4	43.2	55.4	44.1	43.7	74.0	47.5	44.6
`All-pairs`	43.6	29.1	25.5	62.3	34.3	27.1	102.0	48.5	30.2
`SA`	12.3	8.9	9.6	16.3	9.2	9.8	21.5	9.7	10.3

Table 2.7: Application performance on 64 processors

Application	Bandwidths								
	High			Medium			Low		
	rt	ECrt	cfrt	rt	ECrt	cfrt	rt	ECrt	cfrt
`Gauss`	23.8	7.1	4.8	43.4	10.7	5.1	82.8	19.0	5.7
`Matinv`	64.9	62.9	50.3	86.9	76.2	53.6	138.5	109.5	61.1
`Tclosure`	43.2	23.8	23.6	64.6	26.0	24.0	115.2	34.3	24.9
`All-pairs`	66.0	47.1	34.4	103.4	60.4	37.7	182.2	91.7	45.1
`SA`	10.0	4.8	5.4	12.4	5.1	5.5	14.4	5.9	5.8

Table 2.8: Application performance on 128 processors

but the running time under eager combining is close to that of the contention-free machine in most cases. In particular, transitive closure performs close to the optimal under eager combining on high- and medium-bandwidth machines, and is within 7% of optimal on the low-bandwidth machine.

Simulated annealing exhibits an anomaly: the running time under eager combining is better than the running time on the idealized, contention-free machine. Under eager combining, writes (including invalidations) take longer to perform. Thus, processors incur many fewer misses on accesses to the global solution under eager combining. Since the program executes for a fixed number of iterations, the change in the number of misses produces an improvement in running time, even though it may adversely affect the search.

The performance improvement under eager combining is even greater on 128 processors, as shown in table 2.8. Eager combining improves the running time of Gaussian elimination by a factor of 4 on all the machines. Other applications are improved by a factor of 2-3 in most cases. Once again, the running time of transitive closure under eager combining is very close to the optimal running time on an idealized, contention-free machine.

The performance improvements under eager combining are due to an increase in the *effective* network and memory bandwidth of the machine. Without eager combining, long queues can develop both at the network interface and at the memory. The running time under eager combining on a low-bandwidth machine (91.7M cycles) is less than the running time without eager combining on the medium-bandwidth machine (103.4M cycles). The other applications based on locks (Gaussian elimination, transitive closure, and simulated annealing) all run faster under eager combining and low bandwidth than on a machine with high bandwidth and no eager combining. In effect, eager combining multiplies the bandwidth of the multiprocessor by a factor of 2-4 by distributing requests more uniformly in the machine.

Some observations are common to the two tables. For example, the relative performance of eager combining improves in comparison to the standard DASH configurations as we decrease bandwidth. However, decreasing bandwidth also causes the performance of eager combining to worsen relative to the optimal running time on an idealized, contention-free machine. This trend simply indicates that the protocol has intrinsic coherence-maintenance costs, and that bandwidth is a scarce resource even under eager combining.

By comparing the results in the two tables, we see that eager combining does not always improve performance as we increase the number of processors. Even when speedup can be achieved in the absence of contention, as in the case of Gaussian elimination, eager combining may require more bandwidth than is available. On low-bandwidth machines with a large number of processors, both the producer's node and the servers may lack sufficient bandwidth to satisfy all requests. Adding more servers relieves the problem at the server nodes, but creates the need for more network bandwidth at the home node. These results suggest that eager combining on large-scale machines may require multi-level trees in order to reduce contention at the servers and home node, although doing so would increase the number of hops for references by clients.

Bandwidth	Num Procs	rt	ECrt	Brt	cfrt
High	64	9.0	7.6	10.4	6.9
	128	23.8	7.1	10.0	4.8
Low	64	37.9	13.3	18.1	8.0
	128	82.8	19.0	21.8	5.7

Table 2.9: Broadcasting vs eager combining for Gaussian elimination.

2.4.2 Comparison with Broadcasting

A common technique for alleviating contention caused by producer/consumer sharing is to use a form of software broadcasting. With *combining trees* [Yew *et al.*, 1987], processors are organized into a tree structure, in which each processor requests the data from its parent node, with the producer at the root. Each parent node combines the requests of its children into a single request to its parent. Under this scheme, coherence is not guaranteed by the hardware, and the pattern of sharing between processors must be easily predictable.

We implemented consumer-driven software broadcasting, wherein the producer sets a flag indicating when data is ready, and the consumers copy the data. In this implementation multiple consumers can overlap time spent in the network, so many copy operations can proceed in parallel. Each tree node contains several broadcast buffers, which allows a parent node to continue producing data before its children consume the data.

Our software broadcasting tree implements the same degree of fan-out at all levels in the tree, since the traffic generated at each level is the same. This is not true of eager combining, where the root node in the tree (the home node) must broadcast the data to the servers **and** satisfy any requests from clients that were forwarded by servers before they received the data. For this reason, the number of servers per block under eager combining is chosen so that each server has more clients than the home node has servers. Thus, for software broadcasting on a 64-processor machine, we use a two-level tree with a fan-out of 8; under eager combining we use 4 servers, each with 16 clients. For software broadcasting on a 128-processor machine, we use a two-level tree with a fan-out of 12; under eager combining we use 8 servers, each with 16 clients.

We compared the running time of Gaussian elimination under eager combining and software broadcasting, while varying the network and memory bandwidth of the machine. We simulated a machine with high network and memory bandwidth, and a machine with low network and memory bandwidth, since these machines represent the best and worst case for eager combining. Under software broadcasting we assume each memory module has a 16-entry queue; under eager combining, requests are rejected when the memory module is busy. The results of these experiments are presented in table 2.9.

The table presents the running time of the application (in millions of cycles) on a machine using the DASH coherence protocol (rt), under eager combining (ECrt), under

software broadcasting (Brt), and on an idealized, contention-free machine (cfrt). As seen in the table, eager combining performs better than software broadcasting in these experiments. There are several reasons for this. Software broadcasting requires more computation, since processors must explicitly perform data copying. In addition, there is more synchronization overhead under software broadcasting, which must synchronize access to the buffers. Most importantly, in a broadcasting tree, a processor at the bottom of the tree cannot trigger a copy operation higher up in the tree; it must wait for its ancestor to copy the data, before it can copy the data. Under eager combining, a read operation by any client causes the data to be multicast to the servers, and also forwarded to the client; no processor is forced to wait for an ancestor in the tree.

2.5 Summary

In this chapter, we examined the performance implications of a non-uniform distribution of memory accesses caused by producer/consumer sharing of data. Using execution-driven simulation, we observed the effect of variations in memory and network bandwidth on performance. At all levels of bandwidth we considered, our application suite exhibited massive performance degradation on large-scale multiprocessors. Both memory and link bandwidth are limiting factors for our applications. We observed that when memory bandwidth and the unidirectional network bandwidth are comparable (as in most "balanced architectures"), a serious bottleneck may develop at the network interface of the nodes.

To address these bandwidth problems, we proposed the eager combining coherence protocol, which is designed to increase effective memory and network bandwidth. Our experimental results show that this protocol can achieve significant improvements in running time performance (as much as a 4-fold improvement), as a result of the increase in effective bandwidth. In some cases, programs running under eager combining achieve better performance than the same program on a machine with 4 times more absolute bandwidth. We also compared eager combining to software broadcasting, and showed that eager combining consistently outperforms software broadcasting on our application programs.

3 Reducing Contention with Software Interleaving

Eager combining uses sophisticated hardware to alleviate the effects of a non-uniform distribution of memory accesses. Several other techniques, such as multi-stage interconnection networks with combining of memory references [Gottlieb *et al.*, 1983; Pfister *et al.*, 1985], interleaved memory [Pfister *et al.*, 1985], and eager sharing [Wittie and Maples, 1989], also assume some form of special hardware support to tackle memory contention. Although these techniques are known to reduce or eliminate memory contention, their associated hardware can be both complex and expensive, and may depend on particular properties of the interconnection network. Thus, general software solutions are an attractive alternative.

Two software techniques for alleviating contention are software combining trees [Yew *et al.*, 1987] and data replication. Software combining trees are analogous to hardware combining networks, and incorporate logarithmic broadcasting. We can also limit memory contention by replicating data across multiple memory modules. By distributing the requests for data evenly among the copies, we can reduce or eliminate memory contention for the original copy.

Each of the techniques described above is general enough to use in any program. However, our investigation of memory contention in programs for solving linear algebra and graph problems suggests that techniques devoted specifically to parallel matrix computations [Ortega and Romine, 1988] can also be very effective at alleviating contention.

In this chapter, we evaluate the effectiveness of memory interleaving implemented in software. This technique is motivated by the observation that memory contention in matrix computations is typically caused by simultaneous access to a single row of the matrix by multiple processors. If matrices are allocated among memories by rows, simultaneous access to any part of a row requires that processors contend for a single memory module. Memory interleaving spreads memory accesses across several memory modules when multiple processors access a single row of the matrix.

We seek to characterize the source and extent of memory contention in SPMD matrix computations, quantify the costs and benefits of software memory interleaving, and evaluate the tradeoffs between software interleaving and logarithmic broadcasting on large shared-memory multiprocessors.

In the following section we describe our simulation infrastructure and example application programs. Section 3.2 presents our simulation results quantifying the impact of memory contention on their performance. In section 3.3 we describe implementations of our example programs based on software interleaving and quantify the effect of our implementation on the latency of remote memory accesses and the running time of our applications. Our most important contributions are presented in section 3.4, where we analyze the costs and benefits of software interleaving, and compare its performance to both row-major allocation and logarithmic broadcasting. We summarize our results in section 3.5.

3.1 Methodology and Workload

We simulate a large-scale multiprocessor (up to 200 processors) based on a multi-stage interconnection network executing our example applications. Our simulations consist of two distinct steps: a trace collection process, and a trace analysis process. The trace-collection step uses Tango [Davis *et al.*, 1991] to simulate a multiprocessor with (infinite) write-back caches. The traces generated by Tango contain the data references that missed in the local cache of each processor, and all synchronization events.

Our analyzer process takes as input an address trace produced by Tango, and simulates execution of the references in the trace on a distributed shared memory multiprocessor. The analyzer respects the synchronization behavior of the application as represented by the synchronization events contained in the trace. We simulate hardware barriers by allowing all synchronizing processors to leave a barrier at the same time. Lock and unlock operations introduce a short execution delay, 5 cycles. Synchronization events are not allowed to cause contention in our model, although they are critical in maintaining the relative timing of events during trace analysis.

In our machine model, a memory module can process only one request at a time. Requests arriving when the module is busy are rejected and must be reissued. Our analyzer measures contention for memory at the page level; thus each 4KB page is treated as a separate memory module to which requests may be directed. We treat each page as a separate memory module so as to simulate an ideal page placement policy in which contention caused by simultaneous accesses to multiple pages does not occur.

Our simulations assume a cache line size of 64 bytes, a fixed network latency of 36 processor cycles, and local memory latency of 10 processor cycles per cache line. In the absence of contention, a remote memory request requires a request message, a reply message, and memory service time, or 82 cycles total. Each request rejected due to contention suffers a 72 cycle penalty, corresponding to an immediate re-issue of the request. Our assumption that network latency is fixed (i.e., there is no network contention) allows us to isolate the effects of memory contention from network contention. This assumption corresponds to a machine architecture where the network bandwidth per node is much larger than the memory bandwidth. In addition, including network contention in our simulations would assign some of the contention we observe to the network rather than the memory, but would not be likely to affect the tradeoffs we

Application	Running Time		
	50 procs	100 procs	200 procs
Gaussian elim	7.4	8.5	15.6
Matrix inversion	26.1	21.7	26.0
Transitive closure	21.7	12.3	9.0
All pairs	43.0	71.3	136.8

Table 3.1: Running time (in millions of cycles) under row-major allocation.

consider here. Our simulation parameters are somewhat optimistic. Throughout the chapter, we present results demonstrating the performance effect of changing each of our parameters.

Our application workload consists four of the parallel programs we considered in the previous chapter: two linear algebra applications (Gaussian elimination and matrix inversion) and two graph algorithms (transitive closure and all-pairs shortest paths). The input to all the applications is a 512 $\times$ 512 matrix, except for all-pairs which takes a 400 $\times$ 400 matrix as input. Synchronization is implemented with locks in Gaussian elimination and transitive closure, while barrier synchronization is used for matrix inversion and all-pairs shortest paths.

3.2 Effects and Source of Contention

3.2.1 The Effects of Memory Contention

Table 3.1 shows how memory contention affects the running time of our applications. For Gaussian elimination and all-pairs shortest paths, memory contention causes the running time to increase with an increase in processors. In fact, moving from 50 to 200 processors increases the running time of these applications by a factor of 2-3, rather than cutting the running time by a factor of 4. The situation is not quite as bleak in the case of matrix inversion, where 100 processors perform slightly better than 50 processors; however, 200 processors perform no better than 50 processors. Transitive closure is the only program that benefits from an increase in processors, although doubling the number of processors from 50 to 100 only improves performance by a factor of 1.8, and multiplying the number of processors by 4 only improves performance by a factor of 2.4. It is important to note that, for the inputs used in our simulations, these programs have good locality of reference and load balancing properties, and achieve good speedup when contention is not considered. Thus, for all of these programs, memory contention is the major obstacle to effective speedup.

The effects of contention are magnified even more if we relax some of our optimistic assumptions. For example, if we double the memory latency to 20 processor cycles, the effect of contention is even more pronounced. On 200 processors, 92% of the misses in Gaussian elimination suffer contention (up from 84%), the average remote reference latency increases to 2910 cycles (up from 1546), and the running time increases to 28.8 M cycles (up from 15.6 M cycles). Similarly, if we keep memory latency at 10 cycles and

reduce the cache line size to 32 bytes, then 90% of the misses in Gaussian elimination suffer contention, the average remote latency increases slightly to 1571 cycles, and the running time increases dramatically to 30.9 M cycles (since we have doubled the number of remote references). If we both double the memory latency and reduce the cache line size to 32 bytes, then the average remote latency increases to 2904 cycles, and the running time increases to 55.2 M cycles. These results suggest that under less optimistic (and perhaps more realistic) assumptions, memory contention is likely to be an extremely serious problem in the large-scale shared-memory machines we consider.

3.2.2 The Source of Memory Contention

From the results presented in the previous section, it is obvious that all of our example programs suffer from memory contention. Our hypothesis was that the major component of the performance degradation observed in our experiments was due to simultaneous access to a single row of the matrix, as opposed to accesses to a single element. We validated this hypothesis with a simple experiment in which we simulated Gaussian elimination on 50 processors, using a matrix that was allocated so that elements within the same row were placed in different pages. This allocation strategy reduced the average remote access latency from 164 cycles to 83 cycles, which is near optimal. This experiment confirms that the memory contention seen in our examples is due primarily to simultaneous access to the elements of a row, all of which reside in one memory module.

We can also see from our examples that synchronization plays an important role in memory contention. All-pairs shortest paths experiences the worst contention by far, in part because our implementation uses barriers to implement the parallel loop. Transitive closure is similar in structure, but we used locks in its implementation. By using barriers in the all-pairs shortest paths program, we force all processes to access the same row at the same time on every iteration of the outermost loop, thereby increasing contention. To confirm the role of barrier synchronization as a root cause of memory contention in all-pairs shortest paths, we implemented the program using locks instead of barriers on 50 processors. The average latency of a remote memory access fell from 2764 cycles to 247 cycles, and the running time decreased from 43M cycles to 14.4M cycles. It is clear from this experiment that barriers exacerbate the problem of memory contention.

We conclude from these experiments that the major source of contention in our application programs is the synchronized access to the elements of a single row of the matrix, all of which reside in a single page (or memory module). Although relaxing synchronization constraints (by replacing barriers with locks) helps to reduce contention, we still observe substantial performance degradation due to contention in large-scale machines. In the next section we consider an alternative data allocation strategy designed to address this problem.

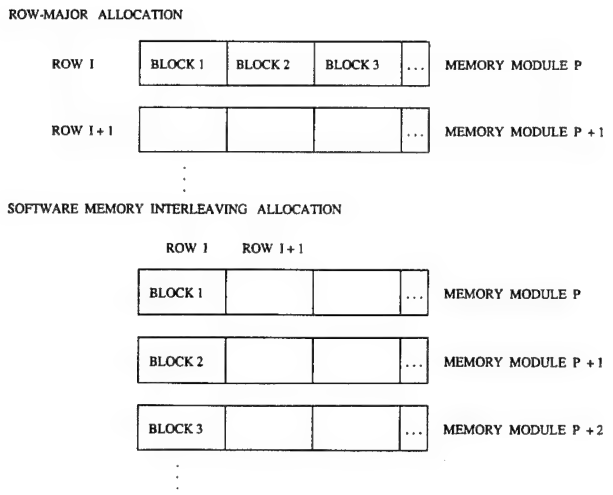


Figure 3.1: Software interleaving matrix allocation.

3.3 Reducing Contention with Software Interleaving

Our experiments in the previous section suggest that the main cause of memory contention in our example programs is the row-major allocation we used for matrices. Row-major allocation places an entire row of the matrix in a single page (or memory module), so that access to the row by multiple processors results in memory contention. Since none of our example programs access a matrix by columns, one obvious way to alleviate memory contention is to allocate the matrices in column-major order. That way, each element of a row resides in a different memory module. However, column-major allocation merely trades memory contention for additional cache misses (due to false sharing), and does not solve the performance problem. We require an allocation strategy that has the spatial locality properties of row-major allocation, and the memory contention properties of column-major allocation. *Software interleaving* has both properties.

3.3.1 Software Interleaving

In software interleaving, we divide each row of the input matrix into cache blocks, and map the cache blocks of a single row into different memory modules. Figure 3.1 shows how matrices are allocated under software interleaving.

Software interleaving is a specific instance of the more general data allocation strategy, referred to as *block scattered decomposition* [Dongarra *et al.*, 1992], in which the size of the block is determined by the architecture's cache line size. In effect, we use column-major allocation of cache blocks, rather than column-major allocation of elements. Since no cache block contains elements from multiple rows, we eliminate the additional cache misses due to false sharing in column-major allocation. Since the cache blocks of a single row map to different memory modules, no memory contention occurs when multiple processors simultaneously access different cache blocks of the same row.

Application	Running Time		
	50 procs	100 procs	200 procs
Gaussian elim	7.7	4.5	2.96
Matrix inversion	25.3	15.3	10.1
Transitive closure	21.3	11.8	6.4
All pairs	15.4	10.5	10.3

Table 3.2: Running time (in millions of cycles) under software interleaving.

Software interleaving can be implemented easily by the compiler, as it only requires the strip-mining loop transformation, and a slightly more complicated addressing of the interleaved data structures. A smart compiler can also determine whether or not to transform a program based on an analysis of its parallel loops. Note that software interleaving avoids modifying the allocation of work to processors and the synchronization determined by the original loop structure.

Software interleaving has a tremendous effect on the average latency of remote accesses observed by our sample parallel programs. For Gaussian elimination, the average remote access latency on 200 processors is 82 cycles, which is optimal. The results for transitive closure are also close to optimal. Average latency for matrix inversion under software interleaving increases slightly with an increase in processors, but still manages a 6-10 fold decrease in average latency when compared with row-major allocation. And even though all-pairs shortest paths still suffers from contention, which results in an average remote access latency of 366 cycles on 200 processors, software interleaving improves the average remote access latency by a factor of 18 to 33.

This decrease in remote access latency produces a corresponding improvement in running time, as seen in Table 3.2. Under software interleaving, each of our applications runs faster with an increase in processors. For Gaussian elimination and transitive closure, doubling the number of processors cuts the running time nearly in half. Additional processors also improve the running time of matrix inversion, although not in the same proportion. Even all-pairs shortest paths continues to exhibit improved running time with an increase in processors, although the performance improvements offered by 200 processors are insignificant. The speedup of matrix inversion and all-pairs shortest paths is limited by the use of barrier synchronization; too many processors waste cycles waiting for a barrier.

Software interleaving is also effective at reducing contention under less optimistic assumptions than those used in the majority of our experiments. For example, even if we double the memory latency to 20 cycles, software interleaving eliminates most memory contention in Gaussian elimination. On 200 processors, only 0.78% of the misses suffer contention, the average latency of remote accesses is only 95 cycles, and the running time only increases by 15%. The same observation applies if we reduce the cache line size to 32 bytes. For Gaussian elimination on 200 processors with a cache line size of 32 bytes, only 0.23% of the remote references suffer from contention, the average remote latency is 82 cycles, and the running time is only 4.0 M cycles. (By way of comparison,

Application	Running Time			
	RM-NC	RM-C	SI-NC	SI-C
Gaussian elim	2.4	15.6	2.7	3.0
Matrix inversion	7.7	26.0	8.7	10.1
Transitive closure	6.1	9.0	6.3	6.4
All pairs	4.0	136.8	4.4	10.3

Table 3.3: Running time (in millions of cycles) with and without contention on 200 processors.

Gaussian elimination under row-major allocation takes 30.9 M cycles on 200 processors when the cache line size is 32 bytes.) If we both double the memory latency and reduce the cache line size to 32 bytes, then only 0.9% of the remote references suffer from contention, the average remote latency rises slightly to 98 cycles (where the minimum is now 92 cycles), and the running time increases to 4.7 M cycles. Thus, the enormous performance advantages of software interleaving are relatively insensitive to memory latency and cache line size.

The conclusion that software interleaving can effectively eliminate the effects of contention holds even if we allocate multiple data rows to a memory module (rather than assign each row of the matrix to a separate page, and treat each page as a memory module). As long as consecutive rows are allocated in different memory modules, there is no significant contention for data within a memory module other than the contention measured in our simulations.

As a final observation, we note that Gaussian elimination runs slightly faster on 50 processors under row-major allocation than under software interleaving. In this case, the additional addressing costs of software interleaving outweigh the benefits associated with reducing memory contention. We will examine those costs in the next section.

3.3.2 Overhead in Software Interleaving

As we discussed earlier, software interleaving transforms the code by applying strip-mining to certain loops. The effect of strip-mining is to replace one loop with two, thereby increasing loop overhead. This overhead is not present when using row-major allocation, and therefore increases the running time of any program using software interleaving, unless offset by a reduction in memory contention.

Table 3.3 illustrates the tradeoff between the overhead associated with software interleaving (SI) and the memory contention associated with row-major allocation (RM). In the table, the NC and C suffixes stand for "no contention" and "contention", respectively. In the absence of memory contention (that is, under the assumption that a memory module can satisfy any number of requests simultaneously), all of our programs execute 3-15% faster on 200 processors using row-major allocation, due to the overhead associated with software interleaving. When memory contention is included, software interleaving clearly dominates, improving performance by an order of magnitude in the

case of all-pairs shortest paths. Recall from Tables 3.1 and 3.2 that software interleaving performs significantly better on 50 processors only for those programs with a large amount of contention (matrix inversion and all-pairs shortest paths). For programs with lower contention levels, software interleaving performs either slightly better (transitive closure) or slightly worse (Gaussian elimination) than row-major allocation on 50 processors. These data suggest that it is not always obvious how to resolve the tradeoffs involved. In the next section we analyze these tradeoffs to determine the circumstances under which to use software interleaving.

3.4 Determining When to Use Software Interleaving

The previous section presented examples of the benefits of software interleaving, and mentioned some of the tradeoffs associated with the technique. This section develops analytical models that explain why software interleaving usually outperforms row-major allocation, and under what circumstances software interleaving outperforms logarithmic broadcasting.

In each case, it is necessary to consider the two kinds of producer-consumer synchronization separately: barrier synchronization and lock synchronization. Under barrier synchronization, we assume that each task begins trying to access a new matrix row immediately after the barrier. This leads to a different analysis from lock synchronization, in which tasks access rows after a lock is set. Under lock synchronization, conflicts in accessing a matrix row are less frequent.

The metric we will use in our comparison is the increase in running time over the optimal case, which has no memory contention and no additional instruction overhead. We measure the running time of the optimal case by simulating the simplest program (row-major allocation) on a system with infinite memory bandwidth (but nonzero memory latency).

Our purpose in performing these analyses is not to develop highly detailed models that can be used to predict the performance of programs. We focus instead on simple models that provide insight into reasons for preferring one technique over another, and that serve as a means of verifying our understanding of the tradeoffs involved.

3.4.1 Modeling Software Interleaving and Row-Major Allocation

For a given cache line size and matrix size, the loop overhead introduced by strip mining is a constant number of cycles. These cycles are distributed among the various processors, and therefore have a decreasing effect on running time as we increase the number of processors. The contention effects under software interleaving depend on the form of synchronization. If processes are loosely synchronized (as is the case when we use locks), then the overhead introduced by software interleaving is almost entirely attributed to loop overhead as follows:

$$SI(P) = \frac{L}{P} + K_1$$

Application	Running Time		
	50 procs	100 procs	200 procs
Opt Gauss	6.5	3.7	2.4
SI Gauss	7.7	4.5	3.0
Opt All pairs	12.4	6.7	4.0
SI All pairs	15.4	10.5	10.3

Table 3.4: Running time of Gauss and All pairs (in millions of cycles) under software interleaving, compared to optimal.

where L is the execution time of the additional instructions introduced by strip mining, and P is the number of processors (assuming good load balance). K_1, which is typically small relative to L, represents the small amount of contention that still occurs under lock synchronization. We find that the quantity K_1 is fixed for each of our programs.

Software interleaving can suffer from memory contention when using barrier synchronization, but only for the first cache line of a row. Subsequent accesses to the same row are skewed by the serial access to the first cache line. The overhead of software interleaving in this case is:

$$SI(P) = \frac{L}{P} + RTP$$

where R is the number of rows in the matrix, and T is the transfer time of a cache line (82 cycles).

As seen in Table 3.4, our experimental results agree with this analysis. For Gaussian elimination, we measure L as approximately 50M cycles and K_1 as approximately 300,000 cycles. For all-pairs shortest paths, we measure L as approximately 70M cycles; from the program, we know that R is 400, and as noted above, $T = 82$. These parameters result in good agreement with the data in all cases.

In contrast, row-major allocation adds no additional loop overhead. However, it suffers serious contention under both barrier and lock synchronization. Under barrier synchronization, all processors contend for the entire row. Since all rows are eventually required by all processors, row-major allocation under barrier synchronization adds overhead equal to the cost of transferring the entire matrix, times P. This is because the last processor to receive a row will get it after $P - 1$ other row transfers have completed. Under barrier synchronization, all the other processors will be forced to wait for the last processor at the next barrier, so all are slowed equally. In other words:

$$RM(P) = \frac{M}{E}TP$$

where M is the number of elements in the entire matrix, and E is the number of elements per cache line.

Under lock synchronization, contention occurs due to random conflicts between processors, as before. However, random conflicts are more common, since processors access a single module repeatedly while transferring a row, and the demand for a particular

Application	Running Time		
	50 procs	100 procs	200 procs
Opt Gauss	6.5	3.7	2.4
RM Gauss	7.4	8.5	15.6
Opt All pairs	12.4	6.7	4.0
RM All pairs	43.0	71.3	136.8

Table 3.5: The running time of Gauss and All pairs (in millions of cycles) under row-major allocation, compared to optimal.

row tends to be greatest immediately after it is produced. In fact, we can determine from the characteristics of our simulated machine that under row-major allocation, it only takes 8 processors transferring rows to saturate a memory module. Since the network trip lasts for 72 cycles, but the memory access itself only takes 10 cycles (which we will call *service time*), no more than 7 consecutive memory accesses can occur during a network trip.

Beyond a certain number of processors, we can expect that at any point in time, at least one memory module is saturated. This observation holds because there are only a fixed number of memories in use; adding more processors adds to the number of requests sent to each memory. The delay caused by a memory module's saturation is eventually propagated to all processes, since each processor (in addition to consuming rows) is producing a row that eventually the other processors will need.

Thus, although it is difficult to model the random contention for memory when the number of processors is small, we can provide an estimate of overhead when the number of processors is large. This estimate is based on the assumption that at any point in time, some module is saturated. We can then see that each additional processor adds an additional service time to the transfer of each cache line, since the additional processor will likely access the module while it is saturated. This means that each additional processor adds the cost of an entire matrix's memory service time, or 10 cycles times the number of cache lines in an entire matrix. So we estimate the overhead of row-major allocation, for large P, and lock synchronization, as:

$$RM(P) = \frac{M}{E} C(P - \theta)$$

where C is the memory's service time (10 cycles), and θ is the threshold number of processors beyond which the system shows memory saturation.

As seen in Table 3.5, our experimental results for row-major allocation generally confirm our analysis. For all-pairs shortest paths, where $M = 400^2$, our predictions are about 30% too high; however, these running times are extremely long and our model predicts them well enough for comparison purposes with software interleaving. For Gaussian elimination, we determine by inspecting the data that memory saturation is reached at about 40 processors, so $\theta = 40$; also, since pivot rows only constitute the upper half of the matrix in Gaussian elimination, $M = 512^2/2$. Our model of overhead for lock synchronization is then quite accurate.

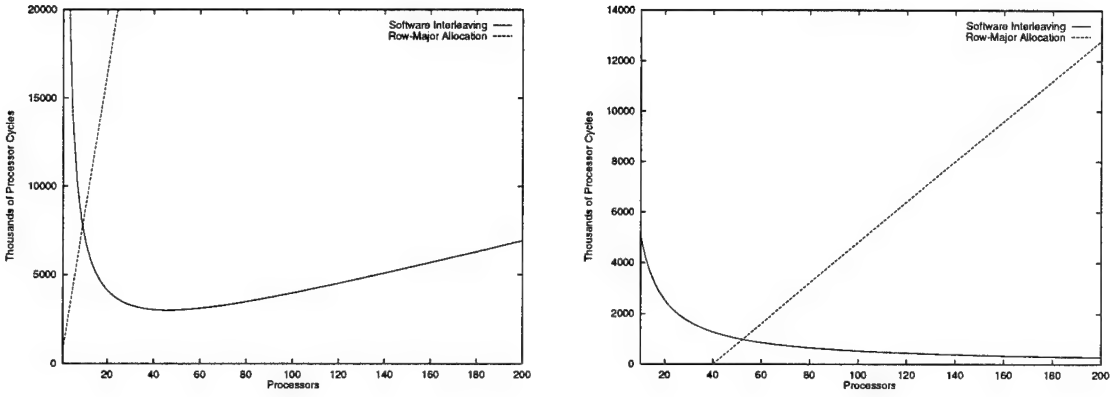


Figure 3.2: Overhead of row-major allocation compared to software interleaving for barrier (left) and lock synchronization (right)

Using this analysis, we can determine when the extra cost of software interleaving is worth paying in exchange for the reduction in contention that it provides. Figure 3.2 shows plots of the analytic models developed above, for the cases of all-pairs shortest paths (on the left) and Gaussian elimination (on the right). The all-pairs graph shows that under the high contention costs of barrier synchronization, software interleaving is preferable even on as few as 10 processors. Beyond about 50 processors, the cost of software interleaving begins to rise, but at a slower rate than the cost of row-major allocation. This trend reflects the difference between contending for the first cache line of the row in the interleaving case, and contending for the entire row in the row-major case.

The analytic models for lock synchronization in Gaussian elimination are plotted on the right side of Figure 3.2. Since contention under lock synchronization starts more slowly than under barriers, more processors are required before software interleaving is preferred over row-major, but the same basic effect is observed: beyond some number of processors (in this case about 50) software interleaving is always preferable.

3.4.2 Comparing Software Interleaving and Logarithmic Broadcasting

The previous section showed that, as the number of processors increases, eventually there comes a point when it is more profitable to use software interleaving over row-major allocation. However, to adequately assess when to use software interleaving, we must compare it to the best known software alternative: logarithmic broadcasting.

We implemented two versions of broadcasting for the row-major Gaussian elimination program. The two versions differ in terms of who drives the broadcast, the producer or the consumers of the data. As our consumer-driven implementation performed significantly better, it is the only one we will present results for.

As pointed out in the last section, 8 processors reading a row can saturate a memory module when the memory latency is 10 cycles and the network latency is 72 cycles; however, as long as the number of processors contending is less than 8, each processor is

Application	Running Time		
	50 procs	100 procs	200 procs
Opt Gauss	6.5	3.7	2.4
Broad Gauss	7.4	4.7	3.4
Opt All pairs	12.4	6.7	4.0
Broad All pairs	15.9	10.3	7.8

Table 3.6: The running time of Gauss and All pairs (in millions of cycles) under logarithmic broadcasting, compared to optimal.

delayed only a small amount. Thus, in our simulated machine, logarithmic broadcasting should not use a tree of degree greater than 8. With this assumption, logarithmic broadcasting can completely eliminate contention when used with lock synchronization. This is because the condition in which some memory module is always saturated does not occur, as it did under simple row-major allocation. Memory modules do not saturate since the complete broadcast of each row is implemented using a much larger set of memory modules, and the number of processors accessing a single module will never be greater than the degree of the tree.

For this reason we can estimate the cost of logarithmic broadcasting under lock synchronization as a constant, which is equal to the extra instructions and synchronization necessary to implement the technique. Thus,

$$LB(P) = K_2$$

where K_2 depends on the specific program. Interestingly, in the programs we studied, K_2 was significant; for example, in Gaussian elimination, $K_2 = 1.0$M cycles. This occurs partly due to the synchronization needed to access broadcast buffers. Ideally each row would have a broadcast buffer on each processor, but that would require expanding the memory usage of the program by a factor of P, which is impractical. Since the amount of buffer space used for row broadcast on each processor must be bounded, buffer space must be re-used, which requires synchronization.

In contrast, under barrier synchronization, the cost of logarithmic broadcasting is not independent of P. The broadcast of each row requires d steps, where $d+1$ is the depth of the broadcast tree.[1] For a tree of degree r, each step requires r row transfers. The first row causes a delay equal to its transfer time; the other rows cause a delay equal only to their memory service times (as discussed earlier in this section). Thus we can estimate the overhead of logarithmic broadcasting under barrier synchronization as:

$$LB(P) = d\frac{M}{E}T + d(r-1)\frac{M}{E}C$$

where d is proportional to $\lceil \log_r P \rceil$.

[1] For a tree of degree r, the depth of the broadcast tree is roughly $\lceil \log_r P \rceil$, although details of how the tree is constructed can change this value by 1 in some cases. In all our experiments, $d = 3$.

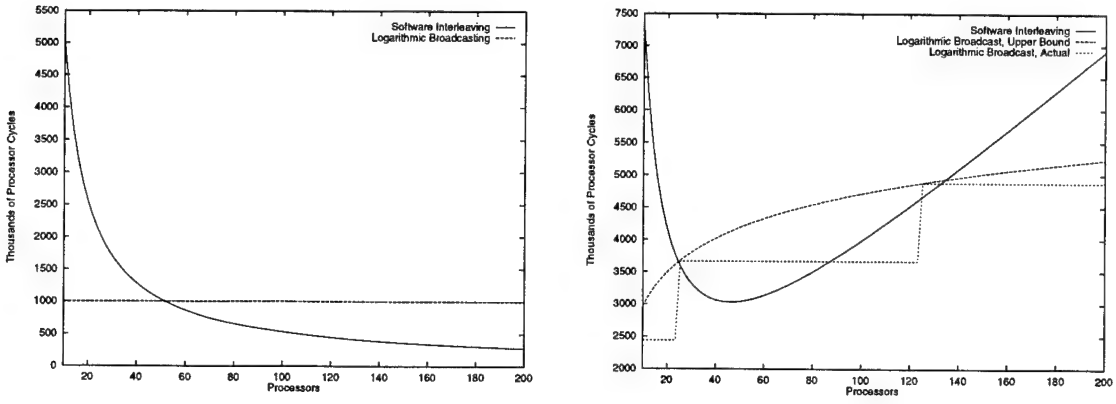


Figure 3.3: Overhead of logarithmic broadcasting compared to software interleaving for lock (left) and barrier synchronization (right)

In our experiments we held d equal to 3, while we varied r to attain the lowest possible value consistent with $d = 3$. For the 50 processor case, we set $r = 4$; for $P = 100$, $r = 5$; and for $P = 200$, $r = 6$. Table 3.6 shows the results of our experiments with All pairs and Gaussian elimination under logarithmic broadcasting, and compares them to their ideal cases. The table shows that $K_2 = 1.0$M cycles is a good estimate of the constant overhead for Gaussian elimination under logarithmic broadcasting. It also shows that our estimate of the overhead due to logarithmic broadcasting under barriers in all-pairs shortest paths is fairly accurate.

Figure 3.3 shows how the two techniques compare. The comparison for lock synchronization is on the left, while the comparison for barrier synchronization is on the right. For lock synchronization, beyond about 50 processors, software interleaving performs better than logarithmic broadcasting. This is because the fixed overhead under software interleaving is lower than that under logarithmic broadcasting. Since contention is much less severe under lock synchronization, the extra cycles required to implement logarithmic broadcasting are more expensive than necessary; software interleaving is preferable due to its simplicity.

The situation is different for barrier synchronization, as shown on the right side of Figure 3.3. This figure shows the overhead of software interleaving compared to logarithmic broadcasting using a tree of fixed degree (equal to 5). The step-function nature of the logarithmic broadcasting curve is due to changes in the depth of the tree as the number of processors increases. The figure also shows an upper bound on logarithmic broadcasting to show that as P grows large, logarithmic broadcasting eventually outperforms software interleaving everywhere. This figure shows that under barrier synchronization contention is so severe that the linearly increasing costs of accessing the first cache line in each row under software interleaving eventually grow larger than the logarithmically increasing costs of broadcast.

Figure 3.3 shows that for large numbers of processors, logarithmic broadcasting is best when using barrier synchronization, but software interleaving is best when using lock synchronization. It also shows that for small numbers of processors, the situa-

tion is reversed: software interleaving is best when using barrier synchronization, while logarithmic broadcasting is best when using lock synchronization.

3.5 Summary

In this chapter we used detailed simulations of application kernels to show that memory contention can substantially degrade the performance of SPMD computations on large-scale shared-memory multiprocessors based on multi-stage interconnection networks. We showed that under row-major allocation, memory contention is due to synchronized access to entire rows of a matrix, rather than simultaneous accesses to isolated data elements. We also showed that software interleaving dramatically reduces memory contention, and therefore performs much better than row-major allocation on large-scale machines.

We analyzed the costs associated with software interleaving and logarithmic broadcasting, and showed how the choice between these two techniques for alleviating memory contention depends both on the type of synchronization used and the number of processors. For large numbers of processors, logarithmic broadcasting is best when using barrier synchronization, but software interleaving is best when using lock synchronization. For small numbers of processors, the situation is reversed: software interleaving is best when using barrier synchronization, while logarithmic broadcasting is best when using lock synchronization. Since the use of barrier synchronization exacerbates memory contention, we conclude that software interleaving and lock-based synchronization is the most effective combination for reducing memory contention in SPMD matrix computations on large-scale machines.

4 Tolerating Remote Access Overhead with Large Cache Blocks

Modern shared-memory multiprocessors use hardware caches to keep data close to the processors that need it, and thereby reduce the average cost of data accesses. The cache block size (that is, the size of coherence and fetching units) is an important design consideration affecting the performance of hardware caches. The choice of block size depends on the locality and sharing properties of applications, as well as the remote access latency and bandwidth of the machine. An application's reference behavior determines the relationship between the block size and the miss rate, while the bandwidth and latency of the machine determine the miss penalty associated with a particular block size, and the performance implications of changes in the miss rate.

In this chapter we examine the effect of block size on the performance of parallel applications in scalable cache-coherent multiprocessors. We are motivated by the observation that an increase in block size may produce a lower miss rate, while an increase in bandwidth lowers the miss penalty for large cache blocks. We would like to pinpoint the upper limits of this argument as it applies to real applications on machines of the foreseeable future.

We first present, in section 4.1, an overview of the tradeoffs involved, including the relationship between block size, miss rate, and miss penalty. We also discuss related work that has examined the issue of block size and its effect on performance.

We use detailed execution-driven simulation of parallel programs on a shared-memory machine to examine the relationship between cache block size and application performance as a function of bandwidth. Our simulation methodology, performance metrics, and application workload are described in detail in section 4.2. In section 4.3 we describe the results of our simulations, which show that block sizes between 32 and 128 bytes provide the best performance for our applications. Larger blocks usually result in an increase in the mean cost per reference, either because the miss rate increases (due to a variety of factors, including but not limited to an increase in false sharing), or because the improvement in the miss rate is not enough to offset the incremental cost of fetching larger blocks.

In section 4.4 we consider whether improving the reference behavior of programs (so as to reduce miss rates) allows us to exploit larger cache blocks effectively. We describe modifications to programs in our application suite that significantly improve the miss rate, and use simulation to determine the optimal block size for these modified

programs. The results of these experiments show that improved locality (and a lower miss rate) does not necessarily lead to an increase in the effective block size. Even if, for a particular program, an improvement in locality allows us to exploit larger cache blocks, subsequent increases in block size are limited by the ever-decreasing benefits of successively larger blocks.

Section 4.5 shows that the observations we made based on our experimental results hold under different assumptions about input and cache sizes.

In section 4.6 we present an analytical model of mean cost per reference, and use it to explain and generalize our simulation results. In particular, we use the model to establish limits on effective increases in block size, and show why these limits apply even in cases where programs exhibit good locality and the architecture provides high remote access bandwidth. We also use our model to show that for the few applications that can effectively exploit very large cache blocks, the overall impact on performance is insignificant unless both the remote access latency and bandwidth are extremely high.

We conclude, in section 4.7, that for most parallel applications, and for the levels of bandwidth and latency we can expect in the foreseeable future, there is little or no performance benefit to using very large cache blocks.

4.1 Application and Architectural Issues Affecting Block Size

Over the last decade there has been little consensus on the choice of block size for coherent caches on shared-memory multiprocessors. The Stanford DASH, MIT Alewife, and Silicon Graphics 4-D series machines all use 16-byte cache blocks. The Berkeley SPUR and NYU Ultracomputer used 32-byte blocks. The Kendall Square KSR-1 and the Stanford FLASH and VMP machines use 128-byte blocks. Paradigm [Cheriton *et al.*, 1991a] supports even longer cache blocks, up to 256 bytes. In part these choices reflect different implementations (e.g., hardware vs. software management of cache misses), but there is clearly a trend towards larger cache blocks for maintaining coherence across processors. Factors that influence the choice of cache block size fall into two categories [Lee *et al.*, 1987]: (1) those that affect the miss rate of applications and (2) those that affect the cost of fetching a cache block.

The spatial and processor (sharing) locality of applications determines how miss rates vary as a function of the block size. Applications with good spatial locality usually benefit from using larger cache blocks, since most of the data in a cache block is likely to be referenced before it is evicted or invalidated. In the absence of write sharing of data, an increase in the block size reduces the miss rate until the *cache pollution* point [Eggers and Katz, 1989]. At that point, useless data begins to replace useful data in the cache, thereby increasing the miss rate.

The relationship between the cache block size and the miss rate has been studied extensively in the context of uniprocessors [Przybylski, 1990; Smith, 1987], but the miss rates of parallel programs do not always follow the same trends as sequential programs [Eggers and Katz, 1989]. Applications with good processor locality (i.e., coarse-grain

sharing) typically favor large cache blocks since, for these applications, the true sharing miss rate goes down with an increase in block size. Applications with poor processor locality (i.e., fine-grain sharing) usually favor small cache blocks, so as to avoid false sharing [Eggers and Jeremiassen, 1991], and to avoid bringing data into the cache that will be invalidated before referenced.

In the best case (perfect spatial locality and coarse-grain sharing), doubling the size of cache blocks would cut the miss rate in half. Unfortunately, this best case scenario is extremely rare; increasing the block size typically causes more misses of one type while reducing the number of misses of another type. For example, if we classify misses as cold start, eviction, true sharing, false sharing, and exclusive request misses (caused by a write to read-shared data), then one can easily see that, as we increase the block size, the number of cold start misses never increases. The number of false sharing misses usually increases as we increase the block size, while the number of misses due to evictions, true sharing, and exclusive requests may decrease initially, but will eventually reach a minimum, and then is likely to increase. As a result of these conflicting trends, an increase in block size may or may not improve the miss rate, depending on the reference behavior of the application, and the structure and size of the cache.

The choice of block size does not depend solely on the miss rates of applications; we must also consider architectural parameters. In particular, remote access latency and bandwidth are important factors, as they determine the cost of fetching a cache block.[1] High remote access latency favors large cache blocks, since more data can be accessed with the same latency penalty. High remote access bandwidth also favors large cache blocks, since more data can be transferred for little extra cost. Large cache blocks can introduce network and memory contention problems however, since small packets generate less contention than large ones (assuming the same amount of data is transferred in both cases) [Agarwal, 1991]. Also, memory performance is affected by the block size; large blocks increase the memory busy time, thereby delaying contending processors.

Increased network and memory bandwidth can reduce the cost of transferring large cache blocks, but do not change the dominant role of the miss rate. An increase in block size only improves performance when the larger blocks result in a lower miss rate. Even then, the decrease in the miss rate must be enough to offset the higher miss penalty associated with larger blocks.

Several researchers have studied the impact of cache block size on the miss rate and overall message traffic on small-scale, bus-based multiprocessors. Agarwal and Gupta [Agarwal and Gupta, 1988] found that 4-byte cache blocks generated the least bus traffic for their application programs. Eggers and Katz [Eggers and Katz, 1989] showed that, for applications with good per-processor locality, increasing the block size from 4 bytes up to 32 bytes improves the miss rate. They also showed that the effect on bus utilization

[1] In this and the following chapters, we refer to the latency of the memory as the time it takes to deliver the first word of data from the memory. We refer to the latency of the network as the time it takes to transfer a single word of data from source to destination. We refer to the bandwidth of the network (or memory) as the number of bytes transferred per cycle after the first word is delivered.

depends on whether the improvement in the miss rate offsets the longer transfer time of larger blocks.

The results of these studies on small-scale, bus-based machines do not apply to scalable, network-based machines, which incorporate very different costs. Although a shared bus offers less communication bandwidth per processor than a direct-connect network, bus-based machines typically have a lower remote memory access latency than network-based machines. Limited bandwidth argues for small cache blocks to avoid contention, while a lower remote access latency reduces the penalty for extra transactions associated with smaller cache blocks. In addition, the broadcasting capability of a shared bus reduces the cost of invalidations, which means that any reductions in invalidation traffic achieved with larger cache blocks are not as significant in bus-based machines as in network-based machines.

In order to determine the feasibility of directory-based coherence schemes in network-based machines, Gupta and Weber [Gupta and Weber, 1992] studied the effect of the block size on the invalidation patterns of parallel programs. They found that data traffic goes up and coherence traffic comes down with an increase in block size, and that overall message traffic is minimized when the block size is 32 bytes. Since this study was mainly concerned with how changes in block size affect message traffic, it did not consider the corresponding effects on the miss rate or mean cost per reference, which have a more direct relationship to application running time. In addition, the argument in favor of 32-byte blocks is based on an assumption of limited bandwidth, since the negative effects of larger blocks are limited to an increase in the number of invalidations per write operation and increased message traffic.

Lee *et al* [Lee *et al.*, 1987] explored the performance effect of different cache block sizes as a function of network bandwidth, both in the presence and absence of explicit data prefetching. Their machine model assumes a multi-stage interconnection network, and a compiler-directed cache coherence scheme. They found that the optimal block size for multiprocessors is much smaller than for uniprocessors, and that explicit data prefetching encourages very small (4-byte) blocks. This study did not consider the dynamic sharing behavior of hardware cache coherence however, and therefore the performance of different cache block sizes on shared writable data could not be observed.

Dubnicki [Dubnicki, 1993] explored the effect of changes in cache block size on the mean cost per reference as a function of latency and bandwidth. He showed that the range of block sizes that minimizes the mean cost per reference of an application suite shifts upward (within the range of block sizes considered) with an increase in network bandwidth. For the particular application suite studied, the range shifted from 16..256 bytes at 20 MB/second to 64..256 bytes at 400 MB/second.

Dubnicki's work used trace-driven simulation, with traces collected on an 8-processor machine. We would expect such small-scale parallelism to result in less sharing and better locality than we would see on a large-scale machine, thereby favoring larger cache blocks. Also, his study assumed infinite caches and did not consider the effects of network contention, again favoring larger blocks. Since this study did not consider blocks larger than 256 bytes, the cumulative effect of these assumptions cannot be measured; no upper bound on effective block size is shown by this work.

None of these earlier studies definitively addresses the issue of block size on scalable shared-memory machines. There are many complex factors (including the miss rate of applications, the cache size, and the latency and bandwidth of the machine), and previous studies either ignore one or more important factors, or assume a different architecture with very different costs. In addition, the results of earlier studies depend greatly on the application workload; specific results regarding block size would have to be reevaluated in light of programming systems and programming techniques that improve locality. Furthermore, no previous study has quantified the performance benefits of very large cache blocks for programs with excellent spatial locality and limited sharing, thereby establishing an upper limit on cache block size in multiprocessors.

In this chapter we examine all of the important factors that influence the choice of block size, consider how improvements in locality affect this choice, and use an analytic model to generalize our results and to quantify the performance benefits of large cache blocks.

4.2 Methodology and Workload

4.2.1 Multiprocessor Simulation

As in chapter 2, we use an on-line, execution-driven simulator that exploits a mixture of interpretation and native execution to simulate unmodified MIPS R3000 object code. We simulate events at the level of processor cycles; all simulation parameters and results are expressed in terms of processor cycles. Our event executor deals with all the major components of a parallel computing system: caches, the interconnection network, local memories, and directories.

We simulate a scalable direct-connected multiprocessor with 64 nodes. Each node in the simulated machine contains a single processor, cache memory, local memory, directory memory, and a network interface. Each processor has a 64 KB direct-mapped write-back cache. The cache block size is a parameter in our study. Caches are kept coherent using an implementation of the DASH protocol with release consistency [Lenoski *et al.*, 1990].

The simulator implements a full-map directory for controlling the state of each block of memory. Each node contains the directory for the memory associated with that node.

Throughout this chapter we will again refer to the ensemble of addressable local memory and directory memory at each node as a "memory module." We simulate memory modules that queue requests (coming either from the cache or network interface) when the module is busy. Memory queues are assumed to be infinite. As should be the case for balanced architectures, we assume that the bandwidth of the memory module is equal to the unidirectional network link bandwidth (which is another parameter in our study). The latency of the memory module is 10 processor cycles.

The characteristics of the practical interconnection networks we simulate are exactly the same as in chapter 2. For comparison purposes, we also implement an idealized,

Level	Path Width	Latency/Switch	Latency/Link	Bi-dir Link Bwidth
Infinite	Infinite	2 cycles	1 cycle	Infinite
Very High	64 bits	2 cycles	1 cycle	1.6 GB/sec
High	32 bits	2 cycles	1 cycle	800 MB/sec
Medium	16 bits	2 cycles	1 cycle	400 MB/sec
Low	8 bits	2 cycles	1 cycle	200 MB/sec

Table 4.1: Network bandwidth levels used in simulated machine.

Level	Latency	Cycles/Word	Memory Bwidth
Infinite	10 cycles	0 cycles	Infinite
Very High	10 cycles	0.5 cycles	800 MB/sec
High	10 cycles	1 cycle	400 MB/sec
Medium	10 cycles	2 cycles	200 MB/sec
Low	10 cycles	4 cycles	100 MB/sec

Table 4.2: Memory bandwidth levels used in simulated machine.

infinite bandwidth network, in which the path width is always larger than the size of messages.

Synchronization events do not generate memory or network traffic in our machine model, although they are used to maintain the relative timing of events.

4.2.2 Performance Metrics

For the most part our focus is on two different metrics: the miss rate and the mean cost per reference. The miss rate is computed solely with respect to shared references. That is, the miss rate is defined as the total number of misses on shared data divided by the total number of references to shared data. We classify misses using the algorithm described in [Dubois *et al.*, 1993] as extended in Appendix A.

The mean cost per reference is defined as the number of each type of reference to shared data (hit or miss) times the average cost (a hit always takes 1 processor cycle to complete) divided by the total number of references to shared data. The mean cost per reference depends on the cost of remote accesses, which in turn depends on the latency and bandwidth of the machine. The levels of bandwidth we use are described in tables 4.1 and 4.2 (based on 100 MHz clocks). As stated earlier, the memory bandwidth is the same as the unidirectional network bandwidth.

4.2.3 Workload

Our application workload consists of six parallel programs: `Mp3d`, `Barnes-Hut`, `Mp3d2`, `Blocked LU`, `Gauss`, and `SOR`. `Mp3d` is a wind-tunnel airflow simulation of 30000 par-

Application	Shared Refs	Shared Reads (% of shared refs)	Shared Writes (% of shared refs)
`Mp3d`	21.1 M	60 %	40 %
`Barnes-Hut`	55.6 M	97 %	3 %
`Mp3d2`	39.3 M	74 %	26 %
`Blocked LU`	47.5 M	89 %	11 %
`Gauss`	64.5 M	66 %	34 %
`SOR`	20.7 M	85 %	15 %

Table 4.3: Memory reference characteristics on 64 processors.

ticles for 20 steps. `Barnes-Hut` is an N-body application that simulates the evolution of 4K bodies under the influence of gravitational forces for 10 time steps. `Mp3d` and `Barnes-Hut` are part of the SPLASH suite [Singh *et al.*, 1992]. `Mp3d2` is a version of `Mp3d` restructured for better cache behavior, as described in [Cheriton *et al.*, 1991b]. `Mp3d2` and `Mp3d` use the same input. `Blocked LU` is an implementation of the blocked right-looking LU decomposition algorithm presented in [Dackland *et al.*, 1992] on a 384 $\times$ 384 matrix. `Gauss` is an unblocked implementation of Gaussian elimination that has been used in other studies, including [LeBlanc, 1988]. We use data replication in order to alleviate contention for pivot rows. The input to `Gauss` is a 400 $\times$ 400 matrix. `SOR` performs the successive over-relaxation of the temperature of a metal sheet represented by two 384 $\times$ 384 matrices. Table 4.3 summarizes the distribution of shared references in our applications on a 64-processor machine.

As is the case with similar studies, simulation constraints prevent experimentation with "real life" input data sets. Simply reducing the input size to manageable levels without changing the cache size could produce unrealistic results however. Therefore the input data sizes used for our applications were chosen in tandem with our choice of cache size. We first determined input sizes that could be simulated in a reasonable amount of time, and then experimented with various cache sizes for those data sets. The cache size we ultimately selected as our default, 64 KB, was chosen so as to avoid too heavy an emphasis on replacement misses; this cache size is the smallest that holds the working set of processors for our applications. We do consider other cache and input sizes however, in order to understand the effect of these parameters on the miss rate and the mean cost per reference of our applications.

4.3 Minimizing the Miss Rate and Mean Cost Per Reference

In this section we explore the effect of changes in block size on the miss rates and mean cost per reference (MCPR) of our application suite. We first use the miss rate to determine the optimal block size for an application under the assumption of infinite

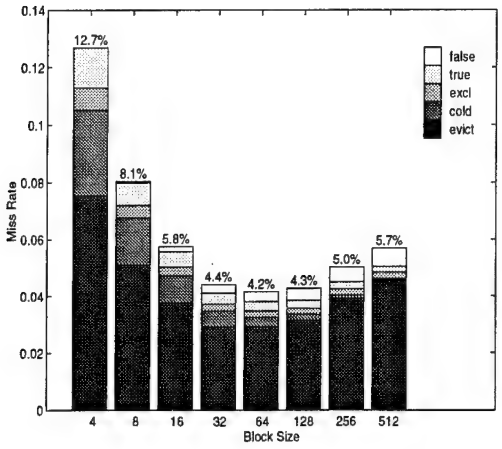


Figure 4.1: Miss rate of **Barnes-Hut**.

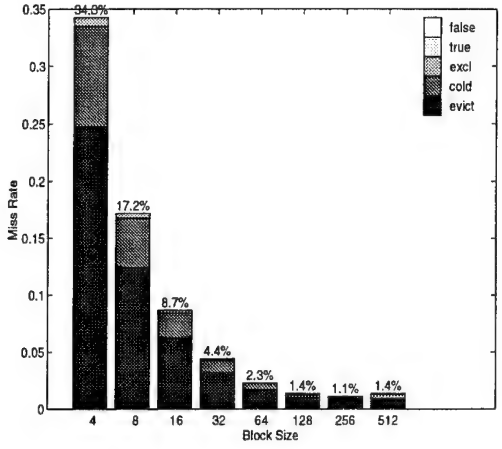


Figure 4.2: Miss rate of **Gauss**.

bandwidth. We then use the MCPR metric to determine the effect of remote access bandwidth and latency on the choice of block size.

4.3.1 Effect of Block Size on the Miss Rate

The block size that results in the minimum miss rate represents an upper bound on the size of cache blocks. Beyond this point, larger blocks simply increase the MCPR (and the running time of the application), regardless of the available bandwidth or the remote access latency. Given infinite bandwidth, the block size that minimizes the miss rate is optimal; smaller blocks incur larger penalties for transferring the same amount of data.

Figures 4.1-4.6 present the miss rates for each of our applications as a function of block size. The percentage at the top of each column represents the percent of all references to shared data that result in a miss; within a column misses are classified as either eviction, cold start, exclusive request, true sharing, or false sharing misses.

Figure 4.1 shows the miss behavior of **Barnes-Hut**. Even though the working set of a processor fits in its cache, the eviction miss rate is still a problem due to limited spatial locality and to the mapping of addresses in direct-mapped caches. The minimum miss rate occurs with 64-byte blocks; larger blocks increase the number of eviction and false sharing misses. The other categories of misses decrease with an increase in block size.

Figure 4.2 shows the miss behavior of **Gauss**. With 4-byte blocks the miss rate is very high (34%), but repeatedly doubling the block size (up through 64 bytes) continually cuts the miss rate roughly in half. The minimum miss rate occurs when the block size is 256 bytes. These improvements in the miss rate are due to the excellent spatial and processor locality of the program. As with **Barnes-Hut**, the miss rate of **Gauss** is dominated by cache evictions. In particular, cache replacements (and exclusive requests) are responsible for the increase in the miss rate when moving from 256 to 512 byte blocks. The high eviction miss rate is due to poor temporal locality in accesses to the

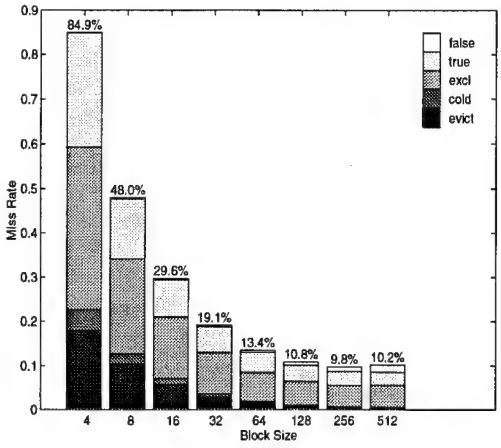


Figure 4.3: Miss rate of `Mp3d`.

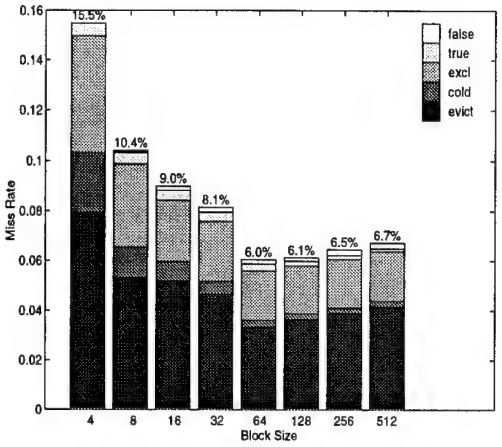


Figure 4.4: Miss rate of `Mp3d2`.

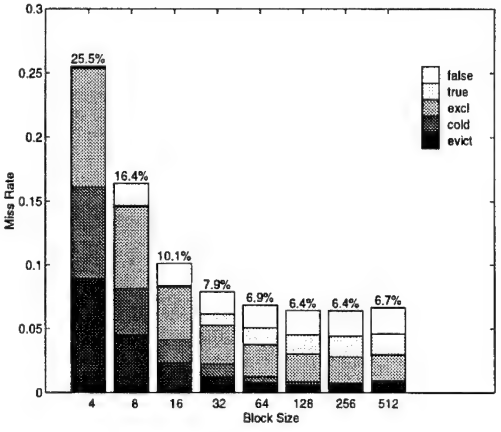


Figure 4.5: Miss rate of `Blocked LU`.

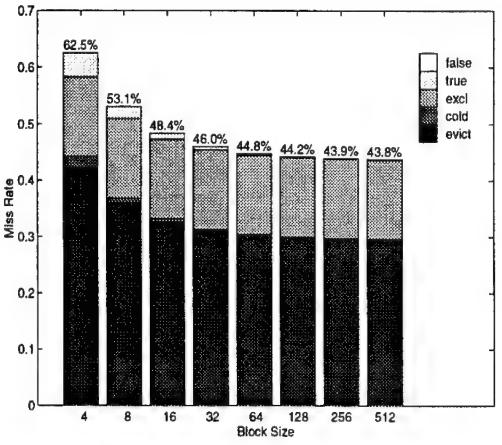


Figure 4.6: Miss rate of `SOR`.

main matrix; each processor repeatedly references a large portion of the matrix for each row it is updating.

As seen in figure 4.3, Mp3d exhibits overall miss rate behavior similar to Gauss. For both programs increasing the block size between 4 and 256 bytes results in a decrease in the miss rate. The composition of the miss rate differs markedly between the two programs however. For Mp3d, false sharing is the limiting factor that precludes the use of 512-byte blocks. The miss rate is high regardless of block size, and in all cases is dominated by sharing-related misses.

Although Mp3d2 is an improvement of Mp3d, the two programs have very different memory referencing and miss rate behaviors. As expected, the miss rates for Mp3d2 are much lower than the corresponding miss rates for Mp3d. It is surprising however that the optimal block size for Mp3d is larger than the optimal block size for Mp3d2 (256 bytes instead of 64 bytes), even though Mp3d2 has much better locality of reference. In the case of Mp3d2, evictions dominate the miss rate, and the number of evictions increases with an increase in block size beyond 64 bytes. This example illustrates why even programs with good locality of reference may not be able to exploit large cache blocks.

Figure 4.5 presents the miss rate behavior of Blocked LU. As in Mp3d, the sharing-related misses dominate the miss rate. For the first time we can see significant amounts of false sharing, which is introduced with 8-byte cache blocks and remains fairly constant with larger cache blocks. Despite the false sharing, the minimum miss rate is achieved with reasonably large cache blocks (128 or 256 bytes).

SOR (figure 4.6) is interesting in that increases in the block size do not have a significant effect on the miss rate, even though the miss rate is extremely high for small cache blocks. Miss rate improvements are limited by the fact that eviction and exclusive request misses are unaffected by increases in block size. The reason for this anomalous behavior is that SOR manipulates two matrices, where the memory size of each matrix is a multiple of the processor cache size. Since each processor modifies the same row indices in both matrices, rows from one matrix collide with the corresponding rows in the other matrix in the direct-mapped cache. In section 4.4 we describe the effects of modifications to SOR designed to eliminate this cache mapping problem.

In summary, for the majority of our applications, the minimum miss rate is achieved by using cache blocks between 64 and 256 bytes in size. Miss rate improvements quickly decline as we approach the block size that minimizes the miss rate for an application. There is no single type of miss that produces an upper bound on cache block size: false sharing, true sharing, exclusive misses, and eviction misses are all significant contributors to the miss rate of some applications and can limit improvements by either driving up the miss rate or remaining fairly constant as we increase the block size. Thus, we can conclude that for these applications, the largest block sizes used in current multiprocessors represent an upper bound on block size; an increase in block size beyond 256 bytes will likely hurt multiprocessor performance regardless of the latency and bandwidth of the machine.

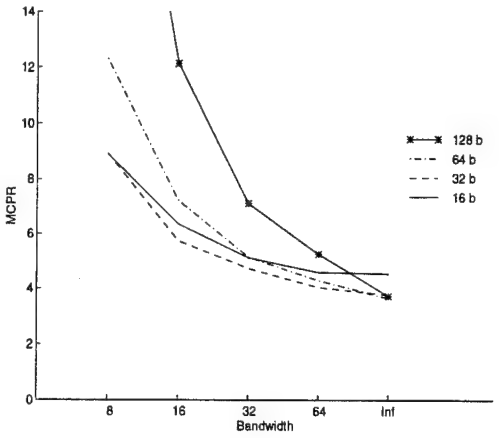


Figure 4.7: MCPR of Barnes-Hut.

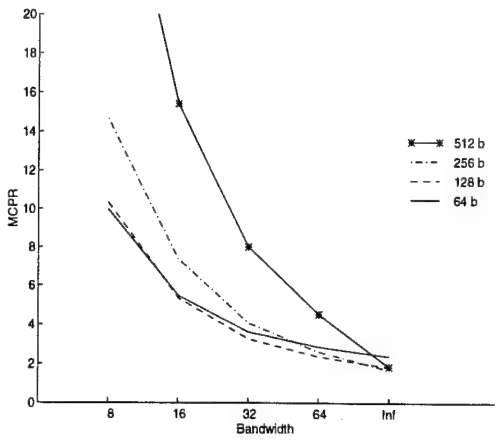


Figure 4.8: MCPR of Gauss.

4.3.2 Effect of Block Size on the Mean Cost Per Reference

In the absence of latency and bandwidth considerations, the miss rate of applications dictates the choice of block size. In practice however, the remote access latency and bandwidth also constrain the choice of block size. Thus, we cannot simply choose the block size that results in the lowest miss rate; we must consider whether any improvement in the miss rate that occurs with an increase in the block size offsets a corresponding increase in the miss penalty, which is dictated by the bandwidth and latency of the machine. We will examine this issue by considering how changes in the block size affect the MCPR.

Figures 4.7-4.12 present the mean cost per reference for our applications, as a function of the block size and the available (network and memory) bandwidth. For each application we only present data for the range of block sizes that results in the lowest MCPR.

Figure 4.7 presents the MCPR for Barnes-Hut. Across a wide range of bandwidth levels, 32-byte cache blocks result in the lowest MCPR. Larger blocks offer competitive performance only at very high levels of bandwidth, even though 64-byte blocks produce the minimum miss rate. At the lowest level of bandwidth, the performance of 16-byte blocks is comparable to the performance of 32-byte blocks. These results suggest that the improvement in the miss rate that occurs when increasing the block size from 16 to 32 bytes (5.8% down to 4.4%) is sufficient to offset the corresponding increase in the miss penalty even at very low bandwidth levels. On the other hand, the improvement in the miss rate that occurs when increasing the block size from 32 to 64 bytes (4.4% down to 4.2%) cannot offset the corresponding increase in the miss penalty, unless infinite bandwidth is available.

The MCPR of Gauss (figure 4.8) and Barnes-Hut exhibit roughly the same behavior. In both cases, a single block size (128 bytes for Gauss vs. 32 bytes for Barnes-Hut) offers the best performance over a wide range of bandwidth levels. Also, this block

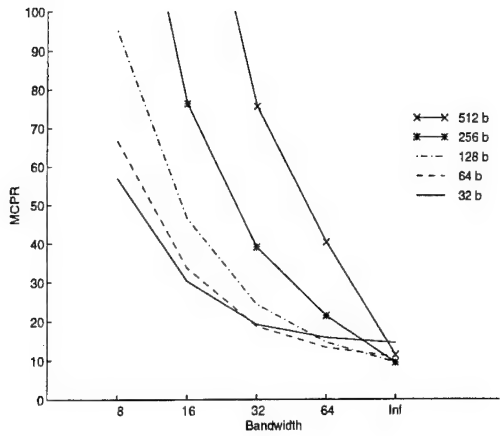


Figure 4.9: MCPR of `Mp3d`.

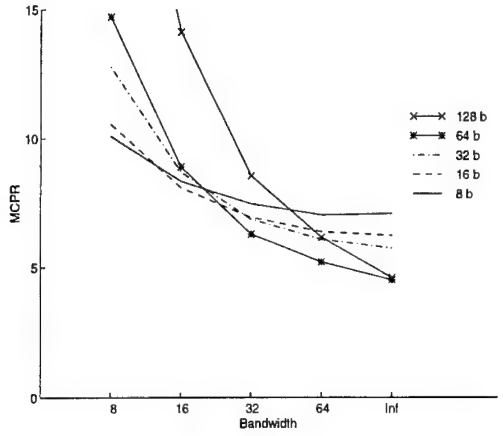


Figure 4.10: MCPR of `Mp3d2`.

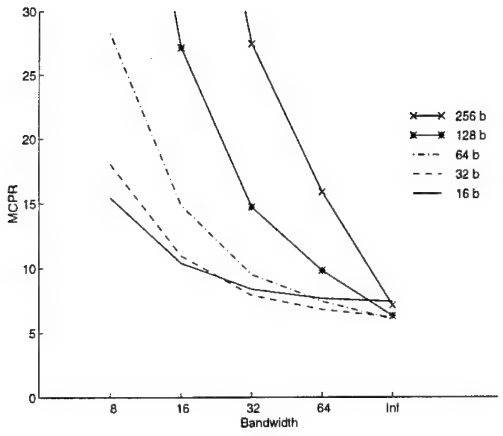


Figure 4.11: MCPR of `Blocked LU`.

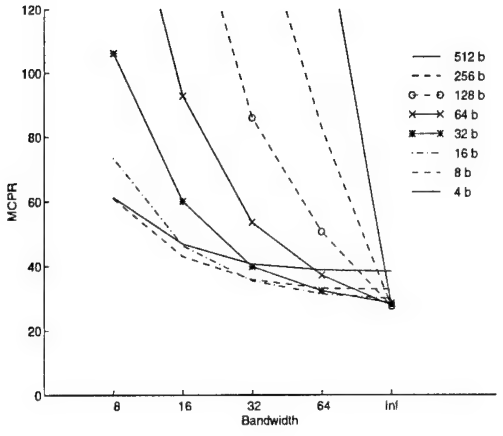


Figure 4.12: MCPR of `SOR`.

size is not the one that minimizes the miss rate (256 bytes for `Gauss` vs. 64 bytes for `Barnes-Hut`).

As seen in figure 4.9, three different block sizes perform best for `Mp3d`, depending on the available bandwidth. At low and medium bandwidth levels, 32-byte blocks perform best, which is quite surprising since the miss rate with 32-byte blocks is almost twice the miss rate with 256-byte blocks (19.1% vs. 9.8%). With high bandwidth, 64-byte cache blocks produce the lowest MCPR. At infinite bandwidth, larger blocks (128 and 256 bytes) prevail.

Figure 4.10 shows a similar trend for `Mp3d2`: small cache blocks (8 bytes) perform best with low bandwidth, slightly larger blocks (16 bytes) perform best with slightly higher bandwidth, and even larger blocks (64 bytes) perform best in all other cases. Also, for the first time, the block size that produces the minimum miss rate also produces the minimum MCPR for practical levels of bandwidth. The reason for this is that the optimal block size (64 bytes) improves the miss rate by 35% over the next smaller block size, which is enough to offset the higher miss penalty associated with the larger blocks. For the other applications the improvement in miss rate between the optimal block size and the next smaller block size is at most 10%, which in most cases is not enough to offset the higher miss penalty.

The best cache block size for `Blocked LU` also depends on the available bandwidth. For machines with low or medium bandwidth, a block size as small as 16 bytes minimizes the MCPR (and therefore the running time), as seen in figure 4.11. For higher levels of bandwidth, the best performance is achieved with a block size of 32 bytes, which is much smaller than the block sizes that minimize the miss rate (128 and 256 bytes). Note that although 128-byte blocks and 256-byte blocks result in the same miss rate, the MCPR of 256-byte blocks is always higher, even under infinite bandwidth. In this particular case, program execution with the larger cache blocks results in more queueing at the memory modules, which significantly increases the miss service time.

Under practical levels of bandwidth, given two block sizes that produce the same miss rate, we would expect the smaller block size to yield a lower MCPR, due to a lower miss penalty. Given infinite bandwidth we would expect the two block sizes to produce comparable MCPRs, except when a change in block size affects the interleaving of remote requests in such a way as to introduce or alleviate memory contention, with a corresponding impact on MCPR.

As seen in figure 4.12, the best cache block size for `SOR` is small independently of the bandwidth available. For machines with low or medium bandwidth, 8-byte blocks perform best, while 16-byte blocks outperform the other block sizes for the other levels of practical bandwidth. The fact that the best-performing block sizes for `SOR` are relatively small is a result of minor fluctuations in miss rate not being able to compensate for the higher miss penalty associated with larger blocks.

Summarizing the results in this section, we showed that several factors contribute to the miss rate of our applications, any one of which can limit effective increases in the block size. In addition, we showed that bandwidth limitations further constrain the size of cache blocks. For most of our applications, block sizes between 32 and 128 bytes provide the best overall performance even under the assumption of relatively high

bandwidth. It is somewhat surprising that no amount of bandwidth suffices to justify blocks much larger than this. In the next section, we consider whether more carefully tuned application programs can exploit larger cache blocks.

4.4 Increasing the Effective Block Size by Improving Locality

The results of the previous section suggest that many shared-memory programs cannot benefit from block sizes larger than 64 or 128 bytes. The question then becomes whether or not locality-enhancing techniques directed at reducing the impact and extent of the dominant class of cache misses in a program would allow for larger cache blocks to be used. In order to investigate this issue, we modified three programs in our application suite so as to alleviate the dominant source of misses in each program.

The first program we modified is SOR. Recall that this program suffers from interference in the mapping of addresses to cache locations; a processor must frequently replace data in the cache, even though the size of its working set is smaller than the size of its cache. To remove this source of eviction misses in SOR we added padding between the two matrices used in the program, so as to ensure that no two rows accessed by a single processor map to overlapping sets of cache blocks. We call the resulting program Padded SOR.

Figure 4.13 shows the miss rate for the modified program. As seen in the figure, our program modification completely eliminates evictions as a source of misses, and thereby dramatically improves the miss rate. As a side effect of reducing evictions, we also eliminated most of the exclusive request transactions required to regain ownership of evicted blocks. As a result, the number of exclusive request transactions is much lower and is now dependent on the block size. Together these effects lower the minimum miss rate from 43.8% to 0.1%, resulting in nearly perfect spatial locality and limited sharing for Padded SOR.

For both SOR and Padded SOR, the minimum miss rate is achieved with 512-byte blocks. Nevertheless, as seen in figure 4.14, under most practical levels of bandwidth, 512-byte blocks produce the lowest MCPR for Padded SOR, while 16-byte blocks produce the lowest MCPR for SOR. For Padded SOR, the substantial improvements in the miss rate that result from increasing the block size up to 512 bytes offset the corresponding increase in the miss penalty, whereas the relatively minor improvements in the miss rate of SOR offered by blocks larger than 16 bytes do not offset any increase in the miss penalty.

Next, we modified Gauss to improve its temporal locality, and thereby reduce the number of eviction misses. We modified the program so that each processor reads a pivot row once, updates all of its local rows based on that pivot row, and then reads the next pivot row. The resulting program is called TGauss.

By comparing the miss rates of Gauss (figure 4.2) and TGauss (figure 4.15) we can see that this modification is very successful at reducing the number of replacement misses. In addition, the overall miss rate of TGauss is a factor of 3 smaller than the miss

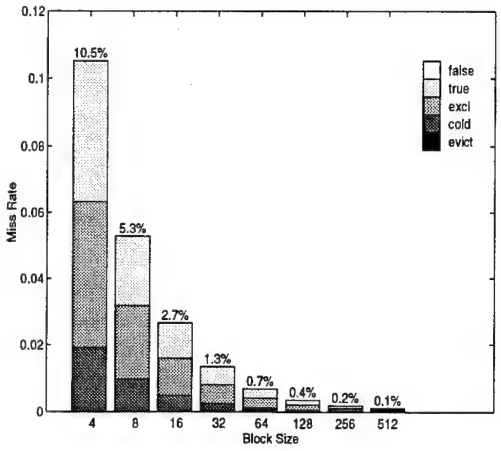


Figure 4.13: Miss rate of `Padded SOR`.

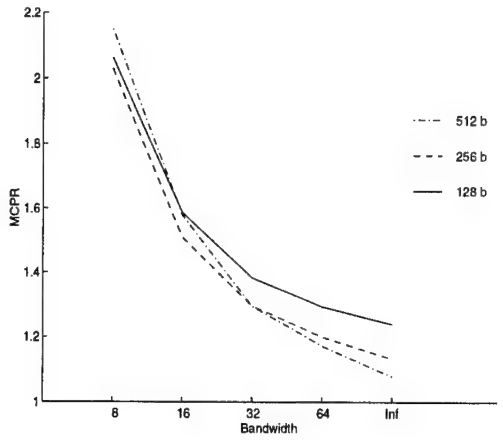


Figure 4.14: MCPR of `Padded SOR`.

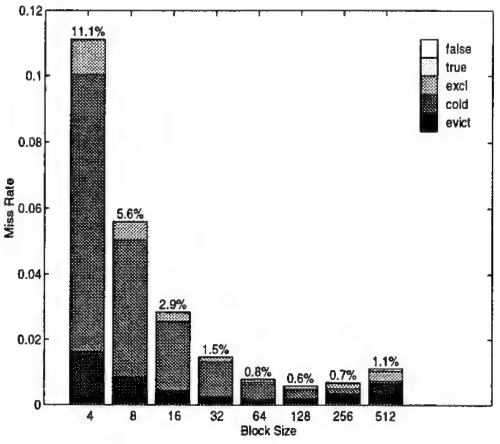


Figure 4.15: Miss rate of `TGauss`.

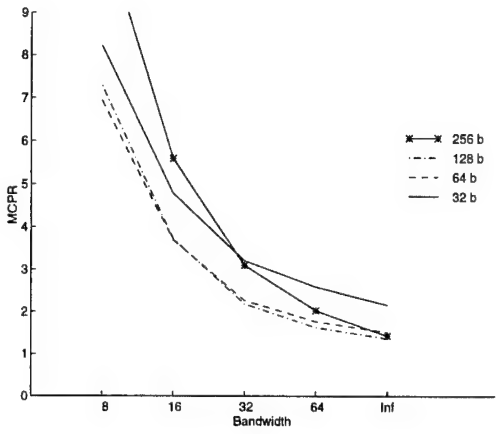


Figure 4.16: MCPR of `TGauss`.

rate of `Gauss` for most block sizes. It is therefore surprising to see that the minimum miss rate for `TGauss` occurs with 128-byte blocks, whereas the minimum miss rate for `Gauss` occurs with 256-byte blocks. The composition of misses is different for the two programs, although evictions are the driving force in the overall miss rate in both cases.

Figure 4.16 shows that even though the upper limit on effective block size for `TGauss` is smaller than the upper limit for `Gauss` (128 vs. 256 bytes), both programs achieve their lowest MCPR with 128-byte cache blocks for all except the lowest level of bandwidth. Thus, in this case, a program modification that improves locality does not increase the size of cache blocks that can be utilized effectively.

Our last program modification involves `Blocked LU`. Recall that the miss rate of this program is dominated by sharing-related misses for block sizes larger than 16 bytes. We modified `Blocked LU` to produce `Ind Blocked LU`, using *indirection* [Eggers and Jeremiassen, 1991] to reduce the number of true, false, and exclusive request misses. We

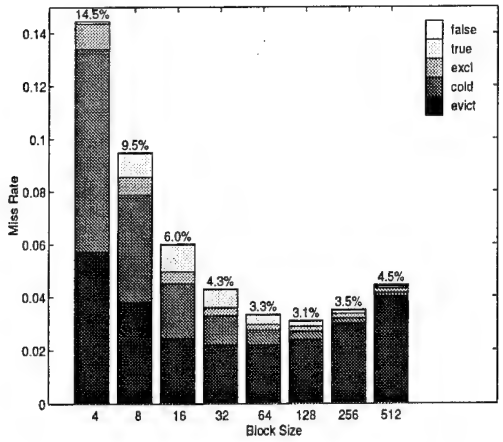


Figure 4.17: Miss rate of `Ind Blocked LU`.

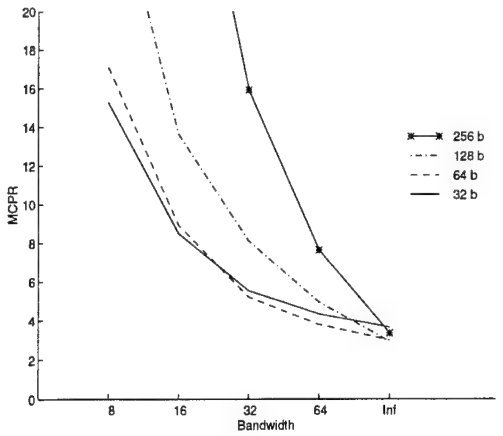


Figure 4.18: MCPR of `Ind Blocked LU`.

added one level of indirection to each access to shared data, and stored the shared data in separate memory regions. Although references to shared data require two memory accesses instead of one (one to read the pointer to the data, and the other to read the data), writes to different shared data locations reference different memory regions and therefore don't conflict. In order for `Ind Blocked LU` to execute faster than `Blocked LU`, the lower miss rate must more than compensate for the additional references (one of which, the reference to the pointer, is usually a cache hit).

The miss behavior of `Ind Blocked LU` is shown in figure 4.17. As expected, the improvement in sharing-related misses is significant, although the number of cold start and eviction misses increases somewhat. The optimal block size is the same for both `Blocked LU` and `Ind Blocked LU` (128 bytes) however. Larger blocks increase the number of evictions, and thereby cause the miss rate of `Ind Blocked LU` to go up. Evictions play a larger role in the miss rate of `Ind Blocked LU` because the use of pointers for indirection effectively increases the working set size of processors.

The MCPR for this application is shown in figure 4.18. Once again, we see that the best block size depends on the available bandwidth. Given low bandwidth, 32-byte blocks outperform all others. For all other levels of bandwidth, 64-byte blocks perform best. Thus, for high levels of bandwidth `Ind Blocked LU` favors a larger block size (64 bytes) than `Blocked LU` (32 bytes).

In summary, we modified three of our programs so as to improve the miss rate with an eye towards increasing the block size. While all of our program modifications were successful at reducing the miss rate, the resulting improvement in locality did not always affect the choice of block size. In two cases (`Padded SOR` and `Ind Blocked LU`) the optimal block size (that is, the block size that results in the minimum miss rate) remained the same, while in the other case (`TGauss`) it actually shrank. The block size that produced the lowest MCPR remained unchanged in one case (`TGauss`), grew slightly in another case (`Ind Blocked LU`), and grew enormously in the third case

(Padded SOR).

These examples clearly show that improvements in locality of reference may not translate to effective increases in block size. In fact, the miss rates of the modified programs are so small that there is little reason to believe that further improvements in locality could help justify larger cache blocks. In section 4.6 we use an analytical model of mean cost per reference to argue that the upper bound on effective block size exhibited by our programs is not likely to be exceeded by other application suites, including those exhibiting good locality and limited sharing. In the next section we consider variations in the cache size and input size simulation parameters, to see whether larger problem sizes or larger caches affect the miss rate in a way that would alter our conclusions about block size.

4.5 Evaluating the Effect of Input and Cache Size

In this section, we evaluate the effect of different input and cache sizes on the choice of block size for our application programs. We first explore input size variations, then cache size variations, and finally draw conclusions based on observations from both sets of experiments.

4.5.1 Varying the Input Size

In many applications, the input size has a direct impact on the miss rate. Increasing the size of the input could increase the replacement miss rate if the working sets become larger than the processor caches. Increasing the input size can also affect the the spatial locality and degree of sharing exhibited by applications, improving the miss rate in some cases. In order to assess the impact of input size on the miss rate, and thus on the choice of block size, we vary the input size for three of our applications: Barnes-Hut, Padded SOR, and TGauss.

We consider three different input sizes for Barnes-Hut: 2K, 4K (the input size used earlier), and 8K bodies. The working sets associated with 2K and 4K bodies fit into our 64KB caches, while 8K bodies produce working sets much larger than the caches. Simulations with these different input sizes show that the overall miss rate behavior of Barnes-Hut is roughly the same across these input sizes, due to the dominating effect of replacement misses. Beyond 64-byte blocks, the replacement miss rate increases, while the combined effect of the other miss categories remains fairly constant. Since the overall miss rate behavior of Barnes-Hut does not change significantly as a function of the input sizes we consider, the MCPR trends we observed for this application are unchanged with variations in input size.

In the case of Barnes-Hut, increasing the input size increases the replacement miss rate (due to an increase in the size of a processor's working set) without significantly improving the locality properties of the application. We expect further increases in input size to result in the same type of behavior.

Padded SOR represents a different class of applications, since an increase in input size does improve the spatial locality and degree of sharing of the program. Increasing the input size from a 384 × 384 matrix (the input size used earlier) to a 512 × 512 matrix increases the working set size from 24KB to 40KB. More importantly however, the minimum miss rate is achieved using 8KB cache blocks, as opposed to the 4KB blocks observed earlier. We also see a change in the MCPR results, where 1KB blocks now achieve the best performance (assuming high bandwidth). Further increases in input size are likely to result in corresponding increases in the effective block size. As we will show in the next section however, such increases in block size may not significantly improve the running time of the program.

We considered three input sizes for TGauss: 600 × 600, 400 × 400 (the input size used earlier), and 200 × 200 matrices of floating point numbers. The working set sizes associated with these inputs vary from about 26KB down to about 4KB. Although increasing the input size does improve the spatial locality of TGauss, while still producing working sets that fit comfortably within the cache, our simulations show that the different input sizes result in very similar miss rate and MCPR behavior. Both the shape of the miss rate curve and the block size that minimizes the miss rate and MCPR remain unchanged. Once again, the increase in the replacement miss rate is not compensated by a decrease in the other types of misses (cold, exclusive request, and true sharing misses), and quickly dominates the overall miss rate behavior for the larger inputs.

In summary, an increase in the input size may not result in an improvement in spatial locality or sharing behavior (as in the case of Barnes-Hut), in which case the choice of block size does not depend on the input size. In some cases (like Padded SOR and TGauss), an increase in the input size does improve the spatial locality and sharing behavior of the program. In such cases, and depending on the cache size, the replacement miss rate may dictate the choice of block size.

4.5.2 Varying the Cache Size

The cache size is an important factor in the choice of block size since, like the input size, it directly affects the miss rate of applications (the replacement miss rate in particular). While larger caches may reduce the replacement miss rate across a range of block sizes, it is unclear whether the block size at which the replacement miss rate begins to increase changes with cache size.

In order to explore the effect of cache size on our results, we varied the cache size for Barnes-Hut, TGauss, and Ind Blocked LU, while maintaining the original input sizes. We observed that the larger the cache, the more likely that an improvement in the other types of misses masks an increase in the replacement miss rate. More generally, as we increase the ratio of the working set size to the cache size, either through an increase in the input size or a decrease in the cache size, we increase the dominance of replacement misses in the overall miss rate trend (as a function of block size). Nonetheless, for each of these applications, the block size at which both the replacement miss rate and the overall miss rate start to increase stays roughly the same, regardless of the cache size and its relationship to the working set size of applications.

All three applications exhibited this same effect; `Ind Blocked LU` is an illustrative example. When using a 32KB cache, the minimum miss rate is achieved with 64-byte blocks, because the increase in the number of replacement misses with 128-byte blocks is not offset by the decrease in the number of cold, exclusive request, and true sharing misses. However, with 64KB or 128KB caches, the minimum miss rate of `Ind Blocked LU` is achieved with 128-byte blocks, because the increase in the replacement miss rate is offset by a decrease in the other types of misses. Nevertheless, for each of these cache sizes, the overall miss rate behavior of `Ind Blocked LU` does not change enough to induce a change in the MCPR results we observed earlier.

In summary, we observed two major effects due to variations in the input size and cache size. First, as a processor's working set grows much larger than its cache, the optimal block size usually decreases due to the greater influence of replacement misses on the overall miss rate. Second, the input size has a significant effect on the choice of block size for those applications that exhibit improved locality with larger inputs. For these applications, the block size producing the minimum miss rate may increase as we increase the input size.

The most important point to note however, is that the trends we observed in our initial experiments, where relatively small cache blocks (up to 128 bytes in size) provided the best overall performance, were also observed for different input sizes and cache sizes. Any improvements in miss rate that derived from improved spatial and processor locality through larger input sizes or larger caches were quickly dominated by an increase in replacement misses for our applications.

4.6 A Model of Mean Cost Per Memory Reference

In this section, we present a simple model of MCPR in k-ary n-cube architectures based on Agarwal's model of network communication [Agarwal, 1991]. Our model relates the miss rate of the application and the remote access latency and bandwidth of the multiprocessor. We first present the model and validate it by comparing the model's predictions to the results in the previous sections. We then use the model to determine the improvements in miss rate required to justify an increase in block size, and to examine the effect of remote access latency on MCPR and the choice of block size.

4.6.1 The Model and Its Validation

We make the following simplifying assumptions in our model:

- Network links are bi-directional and there are no end-around connections.
- Network messages have randomly chosen destinations.
- The probability that a processor initiates a network transaction on any given cycle is uniform across processors.

- Remote requests are satisfied with two-party transactions. That is, a cache miss is satisfied by a single request/reply transaction between the requesting processor and the home node of the requested cache block; no other nodes are ever involved. This assumption is based on our experience with simulations of the DASH coherence protocol which show that two-party transactions dominate.

We define the mean cost per reference to be the number of cache hits times the average cost of a hit plus the number of cache misses times the average miss service time. Thus, the mean cost per reference (MCPR) for a block of size b is:

$$MCPR_b = h_b \times T_h^b + m_b \times T_m^b$$

where h_b is the hit rate with blocks of size b, T_h^b is the average time to service a cache hit (which we assume is 1 cycle), m_b is the miss rate with blocks of size b, and T_m^b is the average miss service time for blocks of size b. To simplify our notation, we will omit the dependence of MCPR, the hit rate, the miss rate, and the service times on the block size in those cases where the block size is fixed.

T_m depends on the time spent in the network, the time spent waiting in the memory queue, and the actual memory service time. More specifically,

$$T_m = 2\left(L_N + \frac{MS}{B_N}\right) + \left(L_M + \frac{DS}{B_M}\right)$$

where L_N and L_M are the average latency at the network and memory (including the average time waiting in the memory queue), B_N and B_M are the path widths (representing bandwidth) of the network and the memory, MS is the average message size, and DS is the average number of bytes provided per request by the memory modules.

The average network latency can be calculated in two ways, depending on whether or not we model contention. In the absence of contention,

$$L_N = D \times T_s + (D-1)T_l$$

where D is the average distance between source and destination, T_l is the message header delay per communication link, and T_s is the delay per switch node. With randomly chosen message destinations, $D = n \times k_d$ for k-ary n-cubes. With bi-directional links and no end-around connections, k_d, the average distance in a single dimension, is $(k - \frac{1}{k})/3$ [Agarwal, 1991].

In the presence of contention, the average network latency is

$$L_N \approx D\left[T_l + T_s + \frac{\rho \times \frac{MS}{B_N}}{(1-\rho)} \frac{(k_d - 1)}{k_d^2}\left(1 + \frac{1}{n}\right)\right]$$

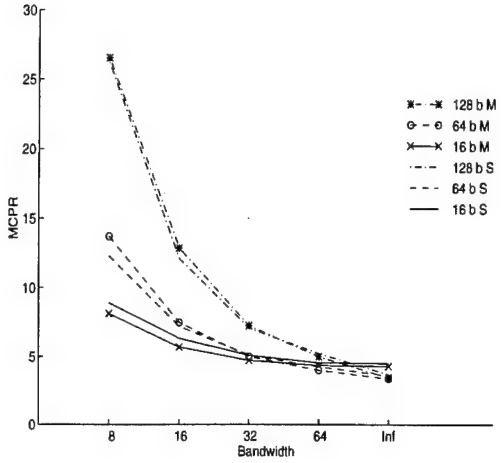


Figure 4.19: Simulated vs. predicted MCPR of `Barnes-Hut`.

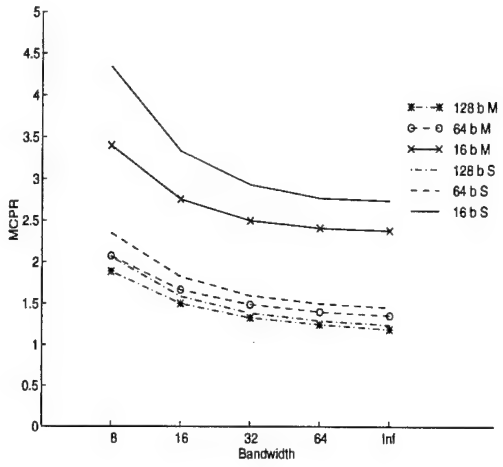


Figure 4.20: Simulated vs. predicted MCPR of `Padded SOR`.

where ρ, the average channel utilization, is $\mu \times \frac{MS}{B_N} \times k_d/2$; and μ, the probability of a network request on any given cycle from a processor, is $\frac{2}{(T_m+(1/m))}$. For further details on the network contention model, see [Agarwal, 1991].

To verify the accuracy of the model, we compare the model's predictions to the detailed simulation results presented in the previous sections. We instantiate the model using data derived from simulations that assume infinite bandwidth. These simulations do not require a detailed cycle-by-cycle simulation of the network, and therefore can be used to provide inputs to the model easily. This approach assumes that the model parameters we collect from simulations with infinite bandwidth (such as the miss rate and the average communication distance) do not change significantly under variations in bandwidth; our experiences with the simulations described earlier suggest this is a valid assumption in most cases.

To instantiate the model, we collect the following statistics from simulations with infinite bandwidth: the miss rate, the average size of network messages, the average service time of the memories (including queue delays), the average number of bytes provided by the memories per operation, and the average distance traveled by network messages. The other architectural parameters used in the simulations are kept constant at the values used earlier: 64 processors, mesh topology, minimum service time of 10 cycles, and a delay per link and per switch node of 1 and 2 cycles, respectively.

We use the statistics from our simulations with infinite bandwidth to instantiate the model and predict the MCPR for a variety of block sizes as a function of bandwidth. We can then compare the MCPR predicted by the model (M) to the MCPR produced via detailed simulations (S). Figures 4.19-4.22 present this comparison for some of our application programs.

As seen in figure 4.19, the model accurately predicts the MCPR for `Barnes-Hut` over a range of block sizes and bandwidth levels. The MCPR predicted by the model is within 10% of the MCPR derived from the detailed simulations for all block sizes and

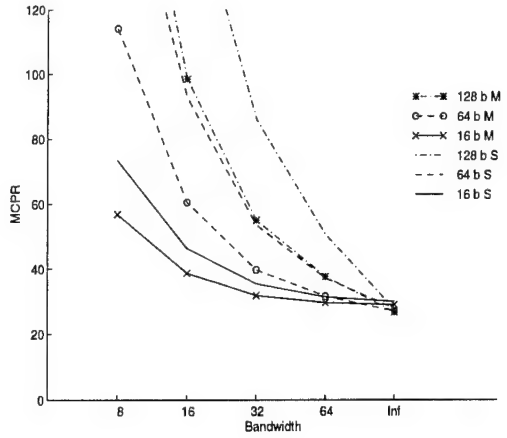


Figure 4.21: Simulated vs. predicted MCPR of SOR.

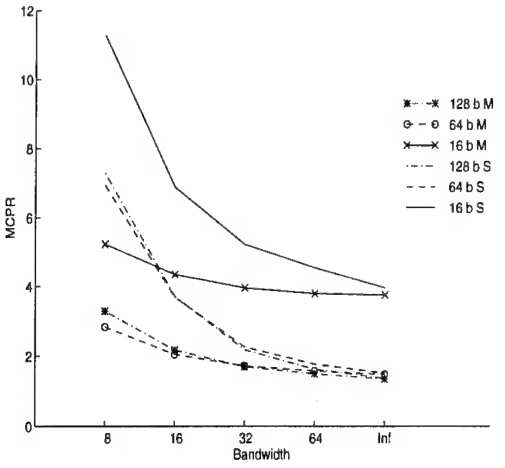


Figure 4.22: Simulated vs. predicted MCPR of TGauss.

bandwidths shown in the figure. The model is just as accurate in predicting MCPR for mp3d2 (not shown), and is almost as accurate for Padded SOR (figure 4.20) and Gauss (not shown), except that the model predictions for Padded SOR with 16-byte cache blocks are 20-30% lower than the MCPRs produced via simulation. Given high bandwidth or small cache blocks the model is accurate for SOR (figure 4.21), mp3d (not shown), and Blocked LU (not shown), but is off by a factor of 2 or more when there is very low bandwidth and large cache blocks. Similarly, the model produces fairly accurate results for TGauss (figure 4.22) and Ind Blocked LU (not shown) with large blocks and high bandwidth, but the predictions for small blocks and low bandwidth are too low by a factor of 2 or 3.

In those cases where the model and simulation results differ, the model fails to accurately account for network and memory contention. Contention may arise any time we have very low bandwidth, which explains why the errors in predictions for Gauss, SOR, and mp3d occur at low bandwidth levels. Contention may also arise when a non-uniform distribution of references results in a hot spot, as occurs in Blocked LU and Ind Blocked LU. Contention can be alleviated by high bandwidth, which explains why all of the predictions at high levels of bandwidth are fairly accurate. Contention may also be alleviated by large cache blocks, if the larger blocks cause a significant reduction in the miss rate (as in the case of Padded SOR and Gauss). On the other hand, larger cache blocks can cause more contention (i.e., SOR and Blocked LU) since network contention increases dramatically with an increase in average message size.

Despite these limitations, the model is fairly accurate when predicting MCPR under high bandwidth, or when the program exhibits a uniform distribution of references. In the remainder of this section we will use the model to predict MCPR under high bandwidth levels.

4.6.2 Quantifying the Benefits of Large Cache Blocks

In the previous sections we claimed that cache blocks larger than 128 bytes are not likely to improve application performance significantly. There are two main reasons for this: larger blocks can increase the miss rate due to sharing behavior or eviction misses, and larger blocks increase the miss penalty without necessarily reducing the number of misses significantly. We will now use our analytic model to show why very large cache blocks are unlikely to improve performance even when programs exhibit good locality and the architecture provides high remote access bandwidth.

We assume that block sizes are a power of two. The evaluation metric we use is MCPR. Thus, we should increase (double) the block size from b to $b \times 2$ only if the MCPR produced by using block size $b \times 2$ is less than the MCPR produced by using block size b. Assuming that the time to service a cache hit is one cycle, the larger block size is preferable when the following holds:

$$(1 - m_{2b}) + m_{2b} \times T_m^{2b} < (1 - m_b) + m_b \times T_m^b$$

where m_i is the miss rate when the block size is of size i, and T_m^i is the time to service a miss for a block of size i. If we assume

$$T_m^b = 2\left(L_N + \frac{MS}{B_N}\right) + \left(L_M + \frac{DS}{B_M}\right),$$

and if we also assume that b is large enough that the message headers are a small percentage of the bytes transferred, and that the proportion of exclusive request misses (in which no real data is transferred) with respect to the total number of misses is roughly maintained when doubling the block size, we can express T_m^{2b} as follows:

$$T_m^{2b} \approx 2\left(L_N + \frac{2 \times MS}{B_N}\right) + \left(L_M + \frac{2 \times DS}{B_M}\right)$$

Doubling the block size improves the MCPR if the improvement in miss rate offsets the increase in miss penalty, which means that the following holds:

$$m_{2b}\left(2\left(L_N + \frac{2 \times MS}{B_N}\right) + \left(L_M + \frac{2 \times DS}{B_M}\right) - 1\right) <$$
$$m_b\left(2\left(L_N + \frac{MS}{B_N}\right) + \left(L_M + \frac{DS}{B_M}\right) - 1\right)$$

Assuming the network and memories have comparable bandwidth (i.e, $B_N = B_M$), this simplifies to

$$m_{2b} < \frac{2MS + DS + B_N(2L_N + L_M - 1)}{4MS + 2DS + B_N(2L_N + L_M - 1)} m_b.$$

When the block size is small, the bandwidth and latency factors dominate, and this ratio is close to 1. Thus, for small block sizes, we need relatively little improvement

in the miss rate to offset the higher miss penalty. As we increase the block size, both MS and DS also increase, and eventually dominate the other factors, at which point the ratio is roughly $\frac{1}{2}$. At that point, doubling the block size must cut the miss rate in half in order to lower the MCPR.

Note that this estimate of the improvement in miss rate needed to justify the next larger block size is conservative, in that it does not take into account any contention caused by using the larger cache block size. To improve the MCPR, the miss rate might have to improve by even more than is suggested by our model.

To illustrate the difficulty of justifying large blocks, consider an application with good locality of reference: `Ind Blocked LU`. Using the statistics collected via simulation under infinite bandwidth (and assuming the architecture is as specified by our simulation parameters), we find that in order to justify an increase in block size from 32 to 64 bytes, the miss rate with 64-byte blocks (3.3%) must be no more than 0.88 times the miss rate with 32-byte blocks (4.3%). Since in this case the improvement in miss rate does compensate for the increase in the miss penalty (assuming high bandwidth), an increase in block size lowers the MCPR (as seen in figure 4.18). A further increase in block size to 128 bytes would not be worthwhile however, since the resulting miss rate (3.1%) is not low enough (2.7% or 0.82 times the miss rate with 64-byte blocks) to justify the increase in the miss penalty.

Even programs with excellent locality may not be able to produce enough improvement in the miss rate to justify large cache blocks. For example, `Padded SOR` has excellent locality; the miss rate for `Padded SOR` decreases with an increase in block size up to 4K bytes. However, even though the miss rate with 1K-byte blocks is very low (0.088%), and is 0.77 times the miss rate with 512-byte blocks (0.114%), it is not low enough; according to the model (and assuming high bandwidth) the ratio must be at most 0.57 to justify an increase in block size from 512 bytes to 1K bytes. Thus, the model correctly predicts that the MCPR with 512-byte blocks is lower than the MCPR with 1K-byte blocks, even though the miss rate with 1K-byte blocks is lower.

Figures 4.23-4.26 show the actual percentage improvement in miss rate as a function of block size compared to the percentage improvement required to offset the higher miss penalty predicted by the model under high bandwidth. These figures illustrate just how hard it is to justify large cache blocks. For `Barnes-Hut` there is a steady *decrease* in the percentage improvement in the miss rate as a function of block size, while a steady *increase* is required to justify large blocks (even under high bandwidth). The trends of ever smaller actual improvements in miss rate and ever larger required improvements eventually cross, even for programs with good spatial locality, such as `Padded SOR` and `TGauss`. The two lines cross at the point at which the improvement in the miss rate associated with larger blocks is not enough to offset the higher miss penalty. The crossover point for `Barnes-Hut` (32 bytes), `Padded SOR` (512 bytes), and `TGauss` (128 bytes) all agree with our detailed simulations.

`Mp3d2` (figure 4.26) is an unusual case in that the percentage improvement in the miss rate gained by moving from 32 to 64 byte blocks is higher than the percentage improvement gained by moving from 16 to 32 byte blocks. Although the actual improvement in the miss rate does not steadily decline, the required improvement does

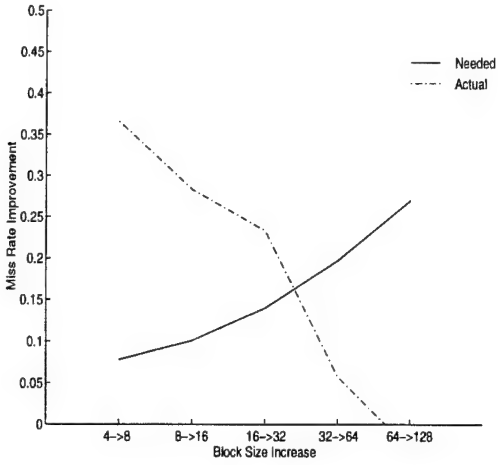


Figure 4.23: Actual vs. required miss rate improvement of `Barnes-Hut`

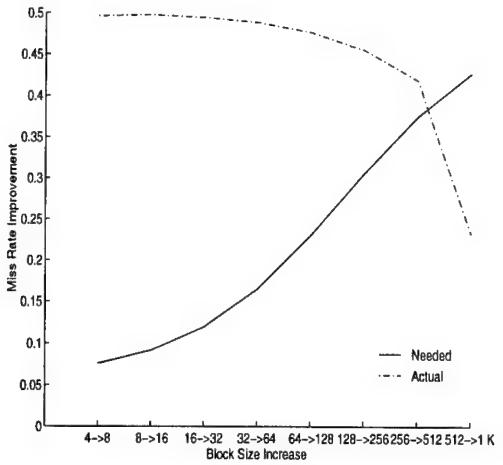


Figure 4.24: Actual vs. required miss rate improvement of `Padded SOR`

steadily rise. Thus, an increase in block size from 8 to 16 bytes is justified, but an increase from 16 to 32 bytes is not. The largest block size for which the actual miss rate improvement is at least the improvement required is 64 bytes, which is consistent with our detailed simulations.

In summary, the improvement in the miss rate required to offset an increase in miss penalty increases with the block size, but this improvement must come from an ever smaller miss rate. For practical levels of bandwidth and for most of our applications, the improvements in miss rate beyond 128-byte blocks are too small to offset the increase in miss penalty.

4.6.3 Implications of High Network Latency

As processor speeds continue to improve, and as we consider building larger and larger multiprocessors, we can expect remote access latency (in terms of processor cycles) to increase. Here we use our analytic model to examine the impact on MCPR of an increase in network latency.

Throughout this chapter, we have assumed that network links impose a 1 cycle delay on each message, while switch nodes impose a 2-cycle delay. We will now consider four levels of network latency: *low* latency assumes delays of 0.5 and 1 cycle for the links and switch nodes respectively; *medium* latency is our original assumption for the delays; *high* latency assumes delays of 2 and 4 cycles respectively; *very high* latency assumes delays of 4 and 8 cycles respectively. Assuming infinite network and memory bandwidth, an average memory latency of 15 cycles, and an average message distance of 6 switch nodes, these latencies roughly correspond to an average remote access latency of 30, 50, 90, and 160 cycles, when moving from low to very high latency.

As a representative example of the effect of network latency on MCPR, consider `Barnes-Hut` with high or very high bandwidth. Recall that 64-byte blocks produce the

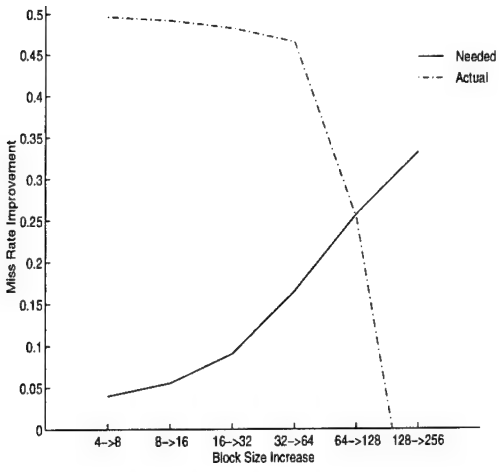


Figure 4.25: Actual vs. required miss rate improvement of `TGauss`

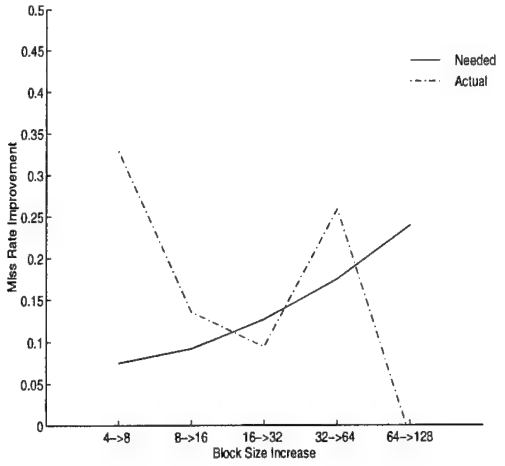


Figure 4.26: Actual vs. required miss rate improvement of `Mp3d2`

minimum miss rate for `Barnes-Hut`, while 32-byte blocks produce the lowest MCPR under our earlier assumptions. As seen in figures 4.27 and 4.28, network latency has a greater impact on MCPR when small cache blocks (8 or 16 bytes) are used, since the small blocks result in a higher miss rate for `Barnes-Hut`, and each extra miss suffers from an increase in network latency. Under high bandwidth, 32-byte blocks produce the lowest MCPR regardless of network latency, although the improvement over 64-byte blocks narrows with an increase in network latency. Under very high bandwidth, the improvement over 64-byte blocks is even smaller, and disappears completely at very high network latency. In fact, for `Barnes-Hut` under very high bandwidth, an increase in network latency from high to very high increases the best block size from 32 to 64 bytes.

Our model for MCPR can be used to explain why the small improvement in miss rate that results from moving to 64-byte cache blocks is enough to offset an increase in the miss penalty under very high bandwidth and very high network latency, but not under other circumstances. Figure 4.29 shows the improvement in miss rate for `Barnes-Hut` needed to justify an increase in block size for our four levels of network latency (assuming high bandwidth). Whatever the network latency, larger block sizes require greater incremental improvement in the miss rate. As seen in the figure, the higher the latency, the smaller the improvement in miss rate required to justify an increase in the block size. This confirms our intuition that large blocks are not as effective with low latency as with high latency, while the reverse is true for small blocks.

Any trend in which network latency (expressed in processor cycles) continues to increase suggests a corresponding trend towards larger block sizes. The upper limit is dictated by the block size that produces the minimum miss rate, with limited bandwidth exerting downward pressure on the block size. Figures 4.30-4.32 illustrate the effects of these trends. Each figure shows the miss rate improvement actually achieved by an increase in block size for an application compared with the miss rate improvement

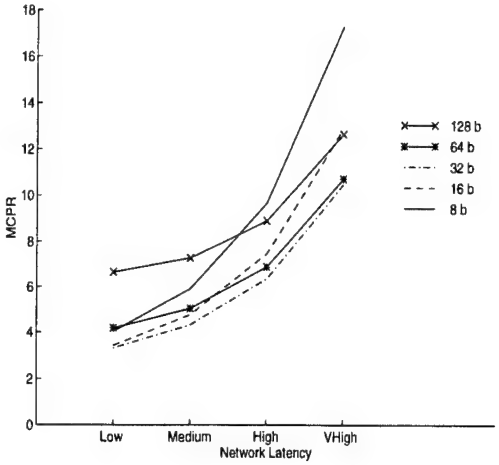


Figure 4.27: Predicted MCPR of `Barnes-Hut` under high bandwidth.

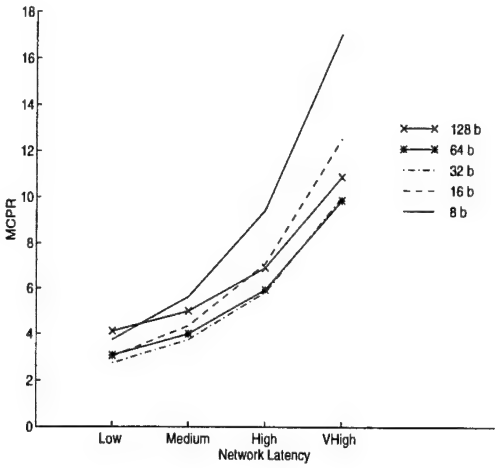


Figure 4.28: Predicted MCPR of `Barnes-Hut` under very high bandwidth.

needed to justify an increase in block size under various combinations of latency and bandwidth. Under every combination of latency and bandwidth, `Barnes-Hut` (figure 4.30) benefits from an increase in block size from 16 to 32 bytes. However, `Barnes-Hut` can exploit 64-byte blocks only on a machine with very high bandwidth and latency, and can never effectively exploit cache blocks larger than 64 bytes (which produce the minimum miss rate). `Mp3d` (figure 4.31) benefits from an increase in block size from 32 to 64 bytes under every scenario of bandwidth and latency, and can effectively exploit a further increase to 128 bytes except for the case of low latency and high bandwidth. 256-byte blocks (which produce the minimum miss rate for `Mp3d`) are only useful under very high latency and bandwidth. 512-byte cache blocks are effective for `Padded SOR` (figure 4.32) under all combinations of latency and bandwidth, but 1K-byte blocks (which produce a lower miss rate) require very high latency and bandwidth to be effective. Given the trends in figure 4.32, even `Padded SOR` is unlikely to be able to effectively utilize cache blocks larger than 1K bytes under most realistic scenarios.

This conclusion holds even if we consider larger problem sizes. For example, if we increase the size of the problem matrix for `Padded SOR` from 384×384 to 512×512, we increase the working set size per processor from 24KB to 40KB. We also increase the block size that minimizes the miss rate from 4KB to 8KB. Nonetheless, the improvement in the miss rate beyond 512-byte blocks is so small that larger blocks do not result in a lower MCPR except in the case of very high latency and high bandwidth.

4.6.4 Achievable Running Time Improvements

The previous sections showed that the miss rate and MCPR of applications rarely improve with cache blocks larger than 128 bytes. However, there are a few programs that do exhibit consistent improvements in miss rate and MCPR as we increase the size of cache blocks. Those programs can be characterized by having extremely good spatial

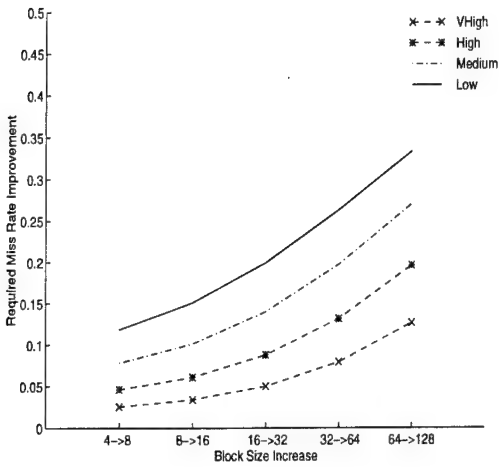


Figure 4.29: Predicted improvement in miss rate required to offset miss penalty for `Barnes-Hut`.

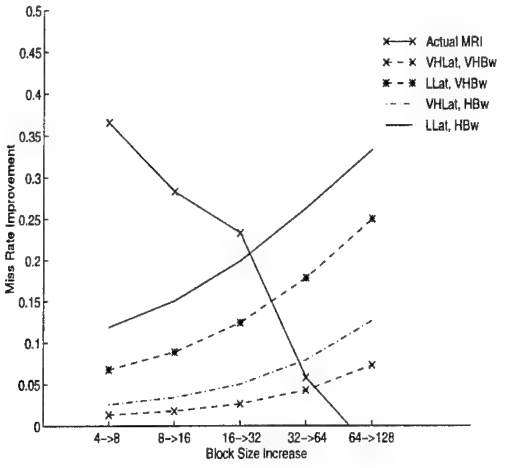


Figure 4.30: Actual improvement in miss rate vs. predicted improvement required for `Barnes-Hut`.

locality and very limited sharing; a good example of this type of application is `Padded SOR`.

We now show that, even for these well-performing applications, the effect of using larger blocks on running time is unlikely to be significant under most practical scenarios. The reason for this claim is that programs with very good locality usually exhibit miss rates so small that, beyond a certain point, any further reduction in the cost of memory accesses has negligible impact on execution time.

Consider for example, the running time improvement achievable by `Padded SOR`. Figure 4.33 presents the maximum running time improvement possible for `Padded SOR` under different levels of memory access latency and assuming infinite bandwidth. "Default" corresponds to our original assumptions about memory and network latencies. The miss rate of `Padded SOR` on an input of size 384×384 is less than 0.12% when the block size is 512 bytes. Given such a small miss rate, the figure shows that it would take an extremely high latency in order for `Padded SOR` to achieve even a 5% improvement in running time beyond 512-byte blocks. Under lower latency assumptions, not even 256-byte blocks provide significant performance improvements, though this block size does cut the miss rate of 128-byte blocks roughly in half. Note that these results are very optimistic, since our infinite bandwidth assumption eliminates any extra costs associated with the larger cache blocks. In practice, performance improvements would be much smaller than those shown in our figure, unless increases in block size are accompanied by matching increases in bandwidth.

To summarize, the block size that minimizes the miss rate is the largest block size worth considering, but the best block size depends on the bandwidth and latency of the machine. Within the range bounded by the smallest possible block size and the block size that minimizes the miss rate, bandwidth limitations argue for a decrease in block size, while high latency argues for an increase in block size. If we assume dramatic increases

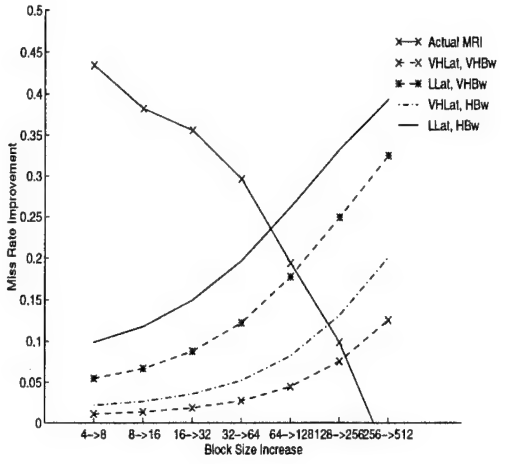


Figure 4.31: Actual improvement in miss rate vs. predicted improvement required for `Mp3d`.

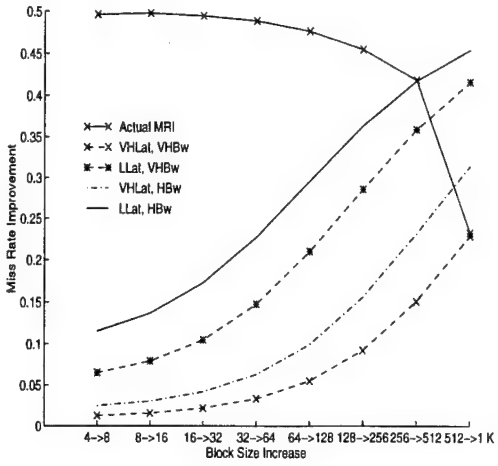


Figure 4.32: Actual improvement in miss rate vs. predicted improvement required for `Padded SOR`.

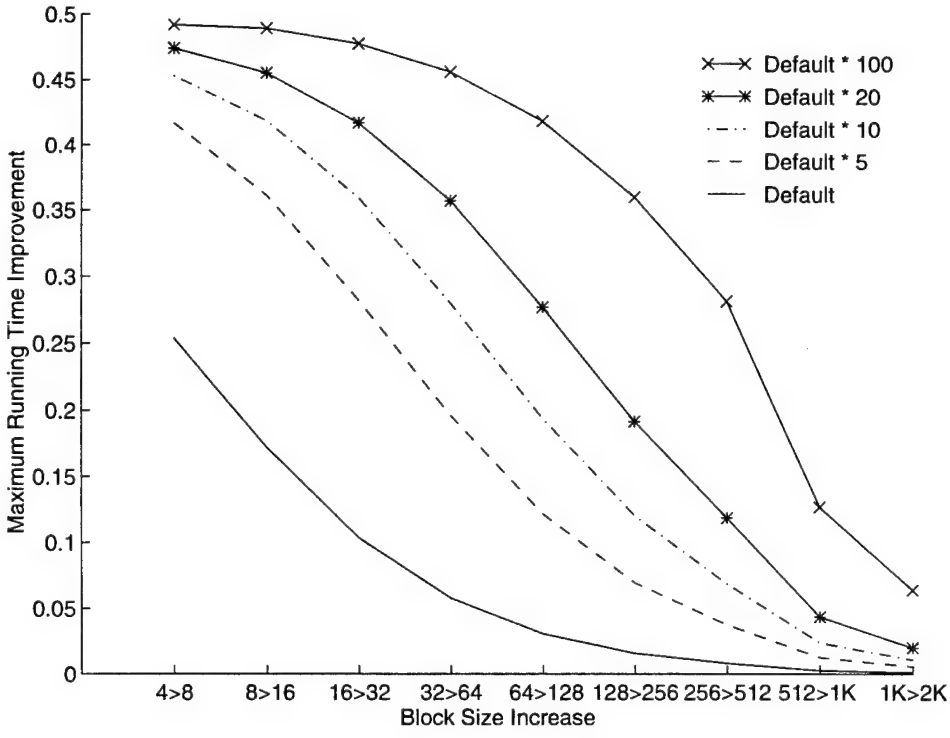


Figure 4.33: Maximum running time improvement achievable by `Padded SOR`.

in both bandwidth and latency, then the best block size will approach the one that minimizes the miss rate, which is rarely larger than 256 bytes. Even if an application has sufficient locality that larger blocks reduce the miss rate, both the latency and bandwidth must be much greater than we will see in the near future if larger blocks are to improve application performance significantly.

4.7 Summary

In this chapter, we examined the relationship between cache block size and application performance as a function of remote access bandwidth and latency. Using execution-driven simulation, we explored the effects of bandwidth on the choice of block size for several parallel programs using the miss rate and mean cost per reference as the primary evaluation metrics. We observed that the reference behavior of applications is such that the block size that minimizes the miss rate usually falls between 64 and 256 bytes. Nevertheless, the block size that produces the lowest mean cost per reference (assuming network latency on the order of 100 cycles and relatively high network bandwidth) usually falls between 32 and 128 bytes. Even in those cases where larger blocks produce a lower miss rate, the incremental improvement in the miss rate due to an increase in block size is often not enough to offset the increase in miss penalty associated with larger blocks.

We were surprised to see that program modifications designed to produce dramatic improvements in locality did not significantly alter our conclusions about block size. In two cases the block size that produced the minimum miss rate remained the same, while in a third case it actually got smaller. The block size that produced the lowest MCPR remained the same in one case, and grew only slightly in another case.

Using an analytical model of mean cost per reference, we showed that the *percentage improvement* in the miss rate required to offset an increase in miss penalty increases with the block size, but this improvement must come from an ever smaller miss rate. Although less improvement is required for higher latency, the actual improvement in the miss rate gained by doubling the block size steadily declines, while the improvement required to offset the miss penalty steadily rises. Our analysis suggests that for most practical levels of bandwidth and latency, and for most parallel applications, the improvements in miss rate beyond 128-byte blocks are too small to offset the increase in miss penalty. Our analytical model also shows that, although the few applications with excellent spatial locality and limited sharing may be able to exploit larger cache blocks, both the remote access latency and bandwidth must be much greater than we can expect in the foreseeable future for large cache blocks to improve performance significantly. We conclude that larger blocks are justified only under extreme circumstances, such as when the remote access bandwidth and latency are both very high, and the majority of applications exhibit nearly perfect locality and very limited sharing.

5 Tolerating Remote Access Overhead with Cache-Miss-Initiated Prefetching

Large cache blocks can be viewed as the simplest example of a data prefetching technique triggered by cache misses. In this chapter we study more sophisticated techniques in this class, *cache-miss-initiated prefetching*, and their ability to tolerate the overhead of remote memory accesses on scalable multiprocessors in the presence of high network and memory bandwidth.

Just like any other prefetching strategy, cache-miss-initiated prefetching techniques are based on an ability to predict, in advance, which addresses an application will reference in the near future. If the predictions are wrong, then the cache is filled with data that will not be referenced soon, resulting in cache pollution. If data is prefetched too early, then it can become stale before it is referenced, requiring a refetch of the data and increasing coherence traffic. Prefetching techniques must balance the benefits of fetching data early with these increased costs.

Two additional characteristics of cache-miss-initiated prefetching techniques are: (1) some cache misses are *required*, since misses provide the only opportunities for prefetching and (2) a large amount of data must be transferred at each miss in order to prevent future misses. These two characteristics of cache-miss-initiated prefetching cause the data traffic of an application to become bursty, since there are fewer misses, but each miss prefetches lots of data. This bursty traffic can result in serious performance degradation, particularly in machines with limited communication or memory bandwidth. Thus, when considering aggressive cache-miss-initiated prefetching, there is an additional tradeoff between lower miss rates and the potential for network and memory contention.

In this chapter we use execution-driven simulation of parallel programs to evaluate these tradeoffs for scalable multiprocessors with high network bandwidth and latency. In particular, we consider the effect on application performance of three different cache-miss-initiated prefetching techniques: (1) large cache blocks, which fetch multiple addresses within a single block, (2) sequential prefetching, which fetches multiple consecutive blocks, and (3) hybrid prefetching, a novel technique combining hardware and software support for stride-directed prefetching.

Our results show that block sizes between 16 and 128 bytes provide the best per-

formance for our applications; larger blocks either increase the miss rate or incur an increase in the miss penalty that dominates any improvement in the miss rate. Our results also show that sequential and hybrid prefetching perform better than prefetching via large cache blocks, and that hybrid prefetching performs at least as well as sequential prefetching. In fact, hybrid prefetching can perform as well as software prefetching, given sufficient bandwidth and regular memory addressing. Based on these results, we conclude that among the cache-miss-initiated prefetching techniques we consider, hybrid prefetching is the only strategy that can offer significant performance improvements for scalable multiprocessors.

The remainder of this chapter is organized as follows. In section 5.1 we describe sequential and hybrid prefetching in detail. In section 5.2 we describe our simulation methodology, performance metrics, and application workload. We present our experimental results in section 5.3, and a summary of our work in section 5.4.

5.1 Overview of Cache-Miss-Initiated Prefetching Techniques

In this section, we overview the tradeoffs involved in each of the cache-miss-initiated prefetching techniques we consider, except for large cache blocks. Refer to chapter 4 for a detailed analysis of the issues involved in the use of that technique.

5.1.1 Sequential Prefetching

Even when large cache blocks reduce the miss rate, their high miss penalty may actually hurt overall performance. One way to reduce the miss penalty, while still retaining the potential for lower miss rates, is to use sequential prefetching with small cache blocks. Under sequential prefetching, a read miss causes some number of successive blocks to be prefetched independently.[1] Prefetches are only issued for blocks for which there are no pending operations, and which are not in the cache at the time of the miss. The processor can continue execution as soon as the block that caused the miss is loaded into the cache. In this way, the cost of prefetching other blocks can be overlapped with computation.

In terms of the read miss rate, sequential prefetching has the potential to perform well for programs that benefit from large cache blocks (i.e., programs with good spatial locality and limited sharing). In contrast to large cache blocks, sequential prefetching is less likely to suffer from false sharing, since the coherence units are small.

In the absence of false sharing, sequential prefetching generates more network transactions than large cache blocks, since several prefetch requests are required to load the data in a large cache block. Each request can be serviced more rapidly however, and

[1] Write misses and requests for exclusive access to shared data could also prefetch additional blocks, but we do not consider these cases. Write buffers and relaxed consistency are sufficient to hide write latencies in most cases.

requests from different processors may get interleaved. This latter feature is particularly important under tight synchronization constraints.

Although sequential prefetching has been studied extensively in the context of uniprocessors (e.g. [Smith, 1978a]), the same is not true for multiprocessors. [Dahlgren *et al.*, 1993] compares the performance of sequential prefetching with an adaptive sequential prefetching technique for scalable multiprocessors. They also studied the performance of large cache blocks that fetch the same amount of data as the sequential prefetching strategy. Their results showed that adaptive prefetching performs at least as well as sequential prefetching, and that both strategies perform better than large cache blocks. This study assumed infinite network bandwidth however, and did not investigate how aggressively prefetches could be issued. In particular, their implementation of sequential prefetching only fetched a single extra block on a read miss.

5.1.2 Hybrid Prefetching

Both large cache blocks and sequential prefetching only work well for programs with very good spatial locality. Stride-directed prefetching [Fu and Patel, 1992], lookahead data prefetching [Baer and Chen, 1991], and [Palacharla and Kessler, 1994] are examples of prefetching techniques that attempt to deal with large strides in data accesses.

Under stride-directed prefetching, a Stride Prediction Table (SPT) is used to store the last memory address referenced by an instruction. Strides are automatically computed by subtracting consecutive memory addresses referenced by an instruction. Once the stride of access for an instruction is computed, a prefetch of the next required memory block can be issued (provided that the stride is non-zero).

Hybrid prefetching is similar to stride-directed prefetching in that both use a hardware table for storing stride-related information. Hybrid prefetching uses an instruction/stride table (IST) indexed by instruction address, where each entry contains the number of blocks to prefetch on a read miss and a stride between the blocks. On a read miss, the cache controller fetches the block that caused the miss and prefetches additional blocks with a certain stride, as determined by the IST entry for the instruction. If the instruction has no corresponding entry in the IST, a single block (with stride 1) is prefetched on its behalf.

Under hybrid prefetching, the compiler computes the stride of access for an instruction and the number of blocks to prefetch on each cache miss, and generates code to fill in the IST. This code usually resides outside loops, and therefore the overhead of changing the table is negligible.

There are several differences between stride-directed prefetching and hybrid prefetching:

- Under stride-directed prefetching, the strides are generated on-the-fly using dynamic reference information, while the stride information is generated by the compiler under hybrid prefetching.
- Stride-directed prefetching only prefetches one block, while hybrid prefetching allows several blocks to be fetched on a read miss.

- Under hybrid prefetching different instructions can prefetch a different number of blocks, while under stride-directed prefetching all instructions prefetch the same number of blocks.

- The IST is managed by the compiler, so the table itself can be very small (i.e., 4 or 8 entries). The SPT requires a separate entry for each prefetching instruction, and therefore must be very large (possibly on the order of 1K entries).

- The SPT resides on the processor chip; the IST resides outside the processor chip, since we only prefetch on cache (read) misses. Thus, under hybrid prefetching, the instruction address must be available outside the processor chip on a read miss.

Like stride-directed prefetching, hybrid prefetching does not perform well with irregular strides. Another drawback of hybrid prefetching is that it depends on the compiler being able to determine strides of access for the relevant instructions in the program. When this analysis is not possible for a particular instruction, hybrid prefetching must either default to a less sophisticated prefetching strategy (e.g., sequential prefetching) or simply avoid prefetching for that instruction.

Hybrid prefetching is strictly more powerful than sequential prefetching, since the IST can be programmed to prefetch blocks with unit stride. In fact, it is easy to resort to sequential prefetching whenever the stride cannot be determined at compile time. In addition, the number of blocks to prefetch on a miss can be varied on a per instruction basis. Hybrid prefetching also compares favorably against large cache blocks, since hybrid prefetching not only performs well for large regular access strides, but also reduces the miss penalty by handling small cache blocks. The main disadvantage of hybrid prefetching is that it requires additional hardware (i.e., the IST).

The extent to which hybrid prefetching dominates other cache-miss-initiated techniques depends on the stride of access in parallel programs. Dahlgren and Stenstrom [Dahlgren and Stenstrom, 1995] found that, for their application suite, strides were often shorter than blocks (assuming 32-byte blocks) and that stride prefetching was not worth its cost. This result *does not* imply that hybrid prefetching is bound to fail; in contrast with stride prefetching, hybrid prefetching has a very low hardware cost, does not need to delay prefetching until a stride is dynamically detected, and can behave exactly like sequential prefetching in cases where strides are short. In section 5.3, we describe the access patterns of our application suite, and evaluate the performance of each of the cache-miss-initiated prefetching strategies under varying assumptions about bandwidth.

5.2 Methodology and Workload

5.2.1 Multiprocessor Simulation

We use the same simulation infrastructure used in chapters 2 and 4. We simulate a scalable direct-connected multiprocessor with 32 nodes. Once again, each node in the

Level	Latency	Cycles/Word	Memory Bwidth
Infinite	24 cycles	0 cycles	Infinite
High	24 cycles	0.5 cycles	800 MB/sec
Medium	24 cycles	2 cycle	200 MB/sec
Low	24 cycles	4 cycles	100 MB/sec

Table 5.1: Memory bandwidth levels used in simulated machine.

Level	Path Width	Lat/Switch	Lat/Link	Bi-dir Link Bwidth
Infinite	Infinite	5 cycles	1 cycle	Infinite
High	128 bits	5 cycles	1 cycle	3.2 GB/sec
Medium	32 bits	5 cycles	1 cycle	800 MB/sec
Low	16 bits	5 cycles	1 cycle	400 MB/sec

Table 5.2: Network bandwidth levels used in simulated machine.

simulated machine contains a single processor, cache memory, local memory, directory memory, and a network interface. The connection between these node components is clocked at half the speed of the processor. Each processor has a 16-entry write buffer and a 128 KB direct-mapped, lock-up free, write-back cache. The cache block size is a parameter in our study. Caches are kept coherent using an implementation of the DASH protocol with release consistency [Lenoski *et al.*, 1990].

As in previous chapters, the simulator implements a full-map directory for controlling the state of each block of memory. Each node contains the directory for the memory associated with that node. In contrast with previous chapters however, here we assume that shared memory is interleaved among the nodes at a memory (cache) block granularity, i.e. consecutive blocks are assigned to successive nodes in round-robin fashion.[2] Memory modules queue requests (coming either from the cache or network interface) when the module is busy. Memory queues are assumed to be infinite. The latency of a memory module (the time it takes a memory module to deliver the first word of data) is 24 processor cycles. The memory transfer rates (after the latency cost) we use are described in table 5.1 (assuming 100 MHz clocks).

The characteristics of the interconnection network we use in this chapter are exactly the same as described in chapter 4, except that switch nodes introduce a 5-cycle delay to the header of each message. The levels of network bandwidth we use are described in table 5.2 (again, assuming 100 MHz clocks).

[2]This memory organization is used in other multiprocessors, including the BBN TC2000 [BBN, 1989].

Application	Shared Refs	Shared Reads (% of shared refs)	Shared Writes (% of shared refs)
`Barnes-Hut`	54.7 M	98 %	2 %
`Gauss`	64.5 M	67 %	33 %
`MMp3d`	12.7 M	64 %	36 %
`Blocked LU`	47.3 M	90 %	10 %

Table 5.3: Memory reference characteristics on 32 processors.

5.2.2 Performance Metrics

For the most part our focus is on three different metrics: the read miss rate, the memory access stall time per processor, and the running time of the application. We ignore write misses in most cases because we assume deep write buffers and release consistency, which serve to hide the cost of writes. The read miss rate is computed solely with respect to shared references. That is, the read miss rate is defined as the total number of read misses on shared data divided by the total number of reads to shared data. We classify misses using an extension of the algorithm in [Dubois *et al.*, 1993] as described in Appendix A.

The memory access stall time (MAST) is defined as the total stall time experienced by all processors due to memory references (read misses and stalls caused by a full write buffer) divided by the number of processors. In most cases, read misses account for almost all of the stall time; unless stated otherwise, write overheads are negligible.

Using running time as a metric accounts for **all** activities that occur during the simulated execution of a program. Accesses to code and private data are modeled as cache hits.

5.2.3 Workload

Our application workload consists of a subset of the parallel programs used in the previous chapter: `Barnes-Hut`, `MMp3d`, `Blocked LU`, and `Gauss`. `Barnes-Hut` simulates the evolution of 4K bodies for 4 time steps. `MMp3d` is an improved version of `Mp3d` and simulates 30000 particles for 20 time steps. In our modified implementation of `Mp3d`, particles are assigned to processors in such a way as to reduce sharing significantly. `Blocked LU` and `Gauss` run on the same inputs as in the previous chapter, 384 $\times$ 384 and 400 $\times$ 400 random matrices, respectively. Table 5.3 summarizes the distribution of shared references in our applications on a 32-processor machine.

5.3 Evaluating Prefetching Strategies

In this section, we evaluate the performance of three cache-miss-initiated prefetching strategies. We first explore the effect of large cache blocks on the read miss rate and

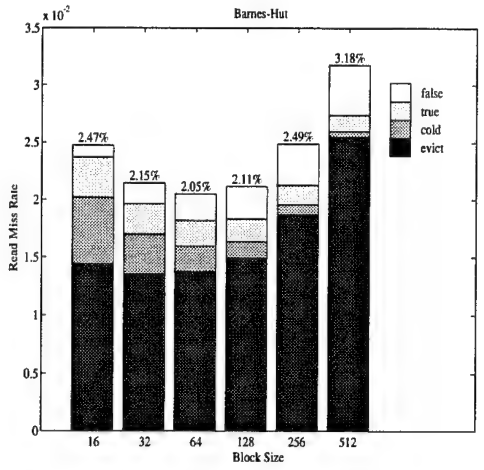


Figure 5.1: Read miss rate of `Barnes-Hut`.

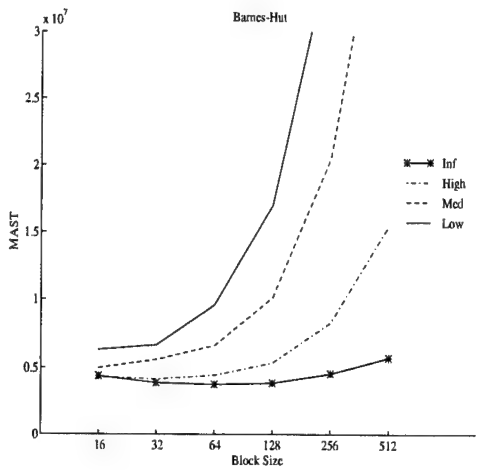


Figure 5.2: MAST of `Barnes-Hut`.

the memory access stall time (MAST) of our application suite. We then investigate the effect of sequential prefetching and hybrid prefetching on the miss rate and MAST as we vary the number of blocks prefetched on a read miss. Finally, we examine the overall effect on running time of each of the cache-miss-initiated prefetching techniques, and compare them to software prefetching, which does not require misses to initiate prefetching.

5.3.1 Large Cache Blocks

Assuming that write buffers and release consistency can hide the cost of writes, then the block size that results in the minimum read miss rate represents an upper bound on the effective size of cache blocks. Larger blocks simply increase the MAST (and consequently the running time) of the application, regardless of the available bandwidth or the remote access latency. Given infinite bandwidth, the block size that minimizes the read miss rate is optimal in terms of the overall remote access cost; smaller blocks incur larger penalties for transferring the same amount of data.

Our ability to hide the cost of writes depends on the block size however. Increasing the block size may increase the cost of write operations (and synchronization latency) due to the resulting higher degree of sharing. Thus, even under infinite bandwidth, the block size that minimizes the read miss rate may not produce the minimum stall time.

Figures 5.1-5.8 present the read miss rates and MASTs for each of our applications as a function of cache block size. In the miss rate figures, the percentage at the top of each column represents the percent of all reads to shared data that result in a miss; within a column misses are classified as either eviction, cold start, true sharing, or false sharing misses.

Figure 5.1 shows the read miss behavior of `Barnes-Hut`. Even though the working set of a processor fits in its cache, evictions are still a problem due to limited spatial locality and to the mapping of addresses in direct-mapped caches. The minimum read

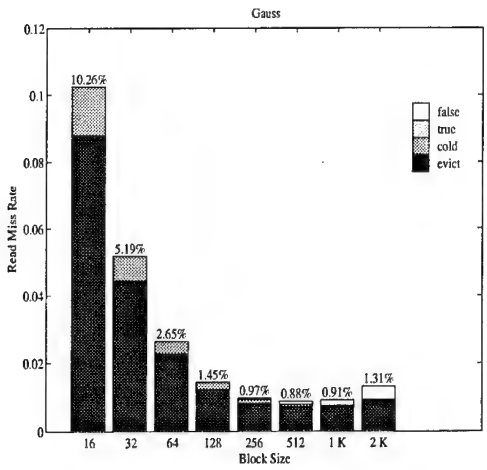


Figure 5.3: Read miss rate of Gauss.

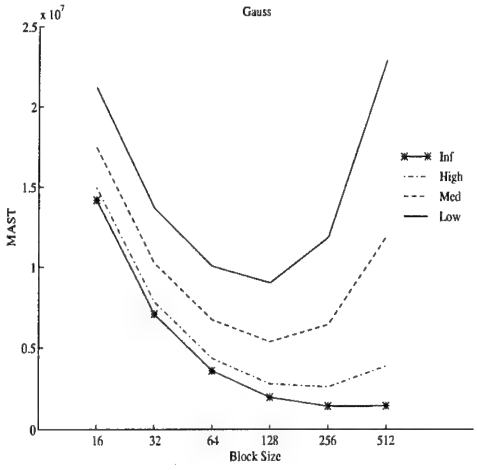


Figure 5.4: MAST of Gauss.

miss rate occurs with 64-byte blocks; larger blocks increase the number of eviction misses (due to cache pollution) and false sharing misses. The other categories of misses decrease with an increase in block size.

Although increasing the block size up to 64 bytes decreases the read miss rate, figure 5.2 shows that the MAST is minimized with 16-byte blocks for most practical levels of bandwidth. The improvements in miss rate for larger blocks are not enough to justify the increased miss penalty. However, under infinite bandwidth 32, 64, and 128-byte blocks perform best; these block sizes offer the lowest read miss rate (around 2.1%) and comparable read miss penalties (around 105 cycles).

Figure 5.3 shows the miss behavior of `Gauss`. As with `Barnes-Hut`, the miss rate of `Gauss` is dominated by cache evictions. Evictions in `Gauss` are caused by poor temporal locality among accesses to the main matrix; each processor repeatedly references a large portion of the matrix for each row it is updating. Repeatedly doubling the block size (up through 128 bytes) continually cuts the miss rate roughly in half. These improvements in the read miss rate are due to the excellent spatial and processor locality of the program. Beyond 128-byte blocks, the read miss rate improves much more slowly, with the minimum miss rate occurring when the block size is 512 bytes. Evictions and false sharing increase the read miss rate when increasing the block size beyond 512 bytes.

Figure 5.4 demonstrates that increasing the block size to 128 bytes significantly reduces the MAST for `Gauss`, regardless of the available bandwidth. However, for the finite levels of bandwidth, increases in the block size beyond 128 bytes do not reduce the MAST, even though the read miss rate is minimized at 512-byte blocks. The read miss penalty for 512-byte blocks is simply too high: 1900, 980, and 320 processor cycles for low, medium, and high bandwidth levels, respectively. 512-byte blocks do perform best with infinite bandwidth, where a small reduction in miss rate is enough to offset a minor increase in miss penalty (to 120 cycles).

As seen in figure 5.5, increasing the block size up to 128 bytes results in a decrease in the read miss rate of `MMp3d`. Although this trend in the miss rate is similar to the

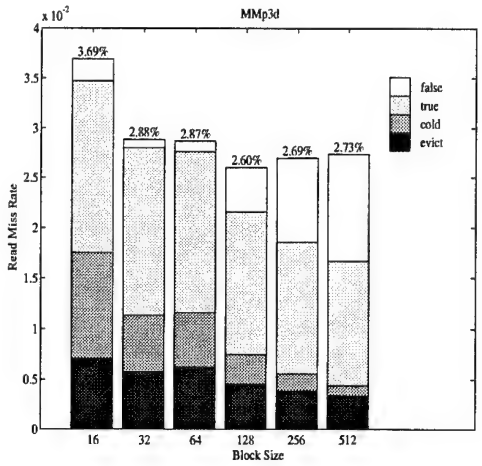


Figure 5.5: Read miss rate of `MMp3d`.

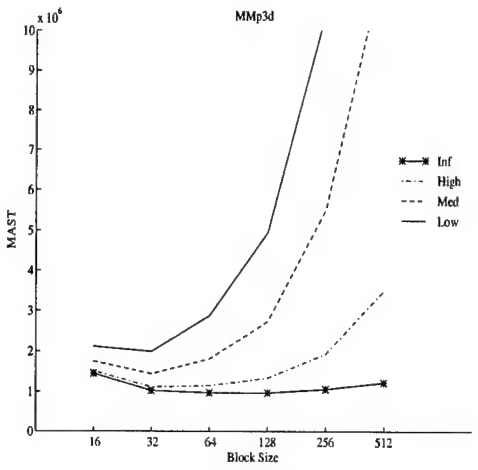


Figure 5.6: MAST of `MMp3d`.

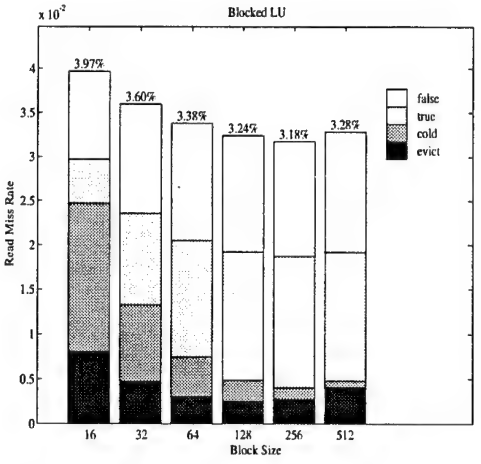


Figure 5.7: Read miss rate of `Blocked LU`.

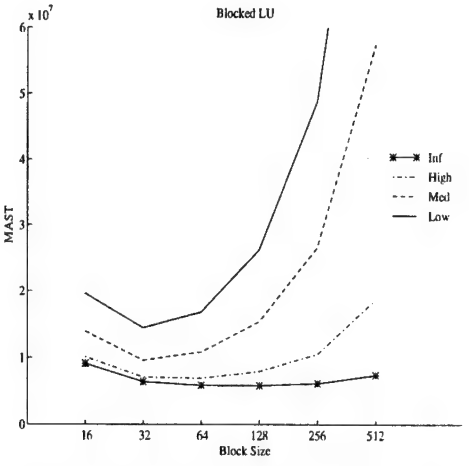


Figure 5.8: MAST of `Blocked LU`.

trends for Barnes-Hut and Gauss, the composition of the miss rate for MMp3d is markedly different. For MMp3d, the read miss rate is dominated by sharing-related misses instead of evictions. False sharing is the limiting factor that precludes the use of 256-byte blocks.

Figure 5.6 presents the MAST of MMp3d. For the low and medium levels of bandwidth, performance suffers when using 128-byte blocks, even though this block size produces the minimum read miss rate. The improvement in read miss rate offered by 128-byte blocks over 64-byte blocks does not offset the increase in the read miss penalty, particularly at lower levels of bandwidth, where miss penalties increase from 245 to 390 cycles under medium bandwidth and from 380 to 690 cycles under low bandwidth. Even with high bandwidth, a fairly small block size (32 bytes) performs as well as, or better than, larger block sizes.

In this case, the excessive memory access stall time produced by large blocks is due primarily to the failure of the write buffer to hide write costs. Large blocks result in more apparent sharing behavior, and hence a greater chance that a write operation will stall the processor. Thus, although writes account for only 10% of the stall time with 128-byte blocks and medium bandwidth, they account for 25% of the stall time with 512-byte blocks and medium bandwidth.

Figure 5.7 presents the miss rate behavior of Blocked LU. As with MMp3d, sharing-related misses dominate the read miss rate when the block size is larger than 16 bytes. For the first time, we see significant amounts of false sharing introduced with relatively small cache blocks. Despite the false sharing, the minimum miss rate is achieved with large cache blocks (256 bytes).

As seen in figure 5.8, Blocked LU and MMp3d have similar MAST behavior. That is, the block size that minimizes the read miss rate (256 bytes) performs much worse than smaller block sizes at the lower levels of bandwidth. Even under infinite bandwidth, most of the performance gains achievable by increasing the block size are captured by a fairly small cache block size (32 bytes).

To see whether more carefully tuned application programs can exploit larger cache blocks, we modified Gauss to improve its temporal locality, and thereby reduce the number of eviction misses. We modified the program so that each processor reads a pivot row once, updates all of its local rows based on that pivot row, and then reads the next pivot row. The resulting program is called TGauss.

By comparing the miss rates of Gauss (figure 5.3) and TGauss (figure 5.9) we can see that this modification is very successful at reducing the number of replacement misses. The overall read miss rate of TGauss is nearly a factor of 6 smaller than the read miss rate of Gauss for most block sizes. It is therefore surprising to see that the minimum read miss rate for TGauss occurs with 128 and 256-byte blocks, whereas the minimum read miss rate for Gauss occurs with 512-byte blocks. The composition of misses is different for the two programs, although evictions are the main driving force in the overall read miss rate in both cases.

Although the upper limit on effective block size for TGauss is smaller than the upper limit for Gauss (128 vs. 512 bytes), both programs achieve their lowest MAST with

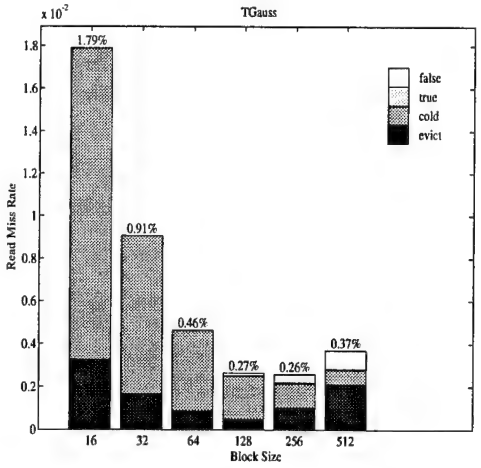


Figure 5.9: Read miss rate of `TGauss`.

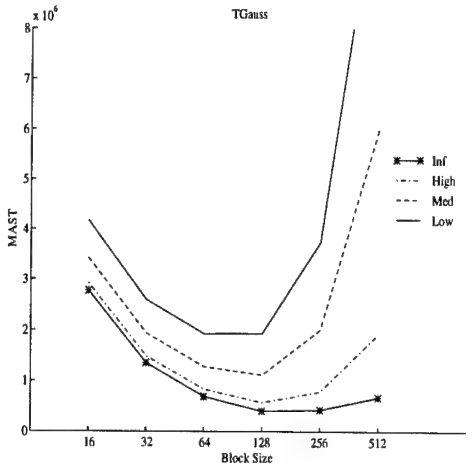


Figure 5.10: MAST of `TGauss`.

128-byte cache blocks in most cases. Thus, in this case, a program modification that significantly improves locality does not increase the size of cache blocks that can be utilized effectively.

From these examples it is clear that several factors contribute to the read miss rate of applications, any one of which can limit effective increases in the block size. Bandwidth limitations further constrain the effective size of cache blocks. For our applications, block sizes between 16 and 128 bytes provide the best MAST under medium and low bandwidth, while block sizes between 32 and 256 bytes perform best with high bandwidth. Fairly small cache blocks (32 or 64 bytes) can achieve most of the performance benefits of larger blocks, even under infinite bandwidth, because larger blocks reduce the miss rate by only a marginal amount. Furthermore, improvements in locality of reference may not translate to effective increases in the block size.

See chapter 4 for a complete and detailed analysis of the effect of block size on the miss rate and application performance.

5.3.2 Sequential Prefetching

In the previous section we saw that increasing the block size can drive up the miss rate or dramatically increase the miss penalty, which precludes the use of large cache blocks as an effective prefetching technique. In this section, we investigate whether sequential prefetching can do better, by alleviating the false sharing and high miss penalties associated with large blocks. Our investigation of sequential prefetching is based on three programs: `TGauss`, `MMp3d`, and `Blocked LU`.

Figures 5.11-5.16 present the read miss rate (under infinite bandwidth) and the MAST of our three applications as a function of the load size (that is, the total number of bytes fetched and prefetched on a read miss) under sequential prefetching. We use 32-byte cache blocks, and vary the number of blocks prefetched on a miss.

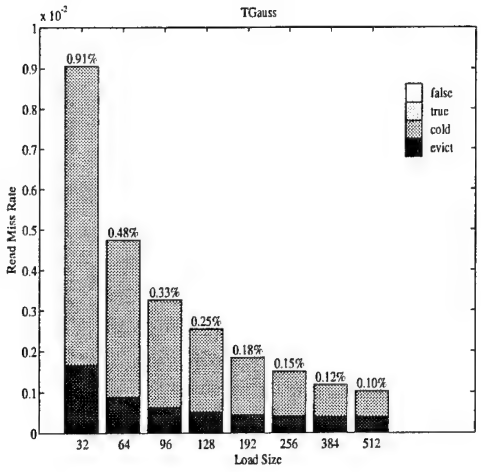


Figure 5.11: Read miss rate of `TGauss` under sequential prefetching.

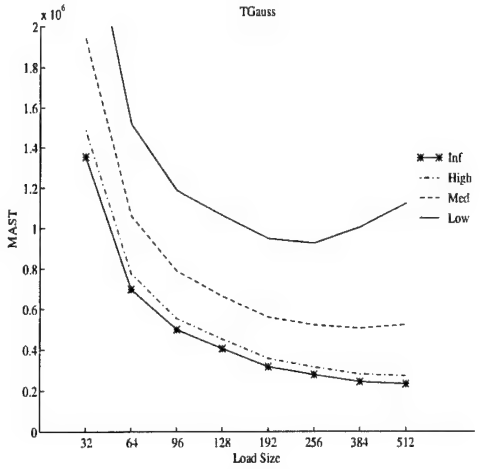


Figure 5.12: MAST of `TGauss` under sequential prefetching.

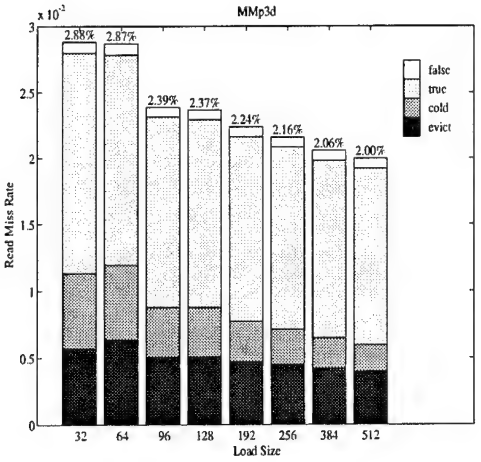


Figure 5.13: Read miss rate of `MMp3d` under sequential prefetching.

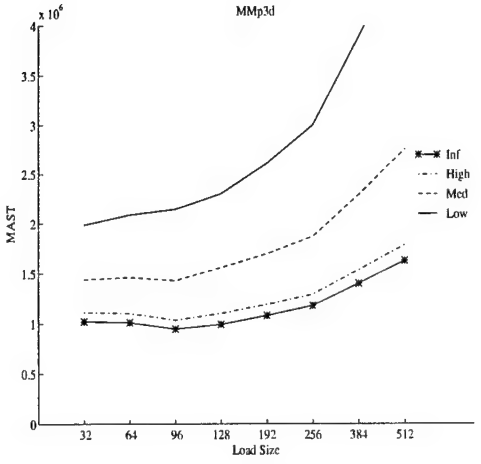


Figure 5.14: MAST of `MMp3d` under sequential prefetching.

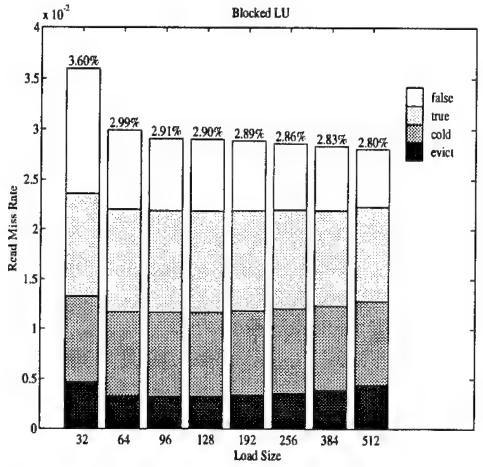


Figure 5.15: Read miss rate of `Blocked LU` under sequential prefetching.

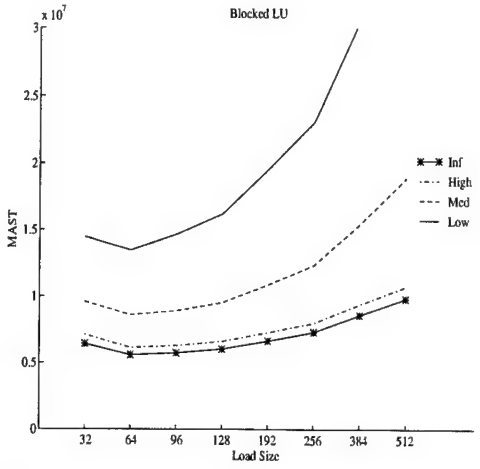


Figure 5.16: MAST of `Blocked LU` under sequential prefetching.

Figure 5.11 shows that sequential prefetching produces lower miss rates than large cache blocks for comparable load sizes. Under sequential prefetching the minimum read miss rate is only 0.10% (with a load size of 512 bytes), which is a factor of 3 smaller than the minimum read miss rate without sequential prefetching. Sequential prefetching performs better with the larger load sizes because it eliminates false sharing misses and reduces the eviction miss rate substantially.

As seen in figure 5.12, sequential prefetching also produces lower stall times than large cache blocks, primarily due to a decrease in the read miss penalty. For example, under sequential prefetching, the read miss penalties for a load of size 128 bytes are 220, 170, and 125 cycles for low, medium, and high bandwidth, respectively. The miss penalties for 128-byte cache blocks are 530, 310, and 160 cycles respectively.

Figure 5.12 also shows that an increase in bandwidth allows more aggressive prefetching. For example, under low bandwidth, 256 bytes is the largest load size that reduces the MAST, but 384 bytes can be effectively utilized given medium or high bandwidth.

As seen in figure 5.13, the read miss rate of `MMp3d` is also reduced by sequential prefetching. The minimum read miss rate is reduced from 2.6% to 2.0%. This improvement in the miss rate is due almost entirely to fewer false sharing misses.

Although larger load sizes reduce the read miss rate under sequential prefetching, figure 5.14 shows that these improvements in the miss rate may not translate to reductions in the stall time. At the lowest level of bandwidth, a load size of 32 bytes produces the lowest MAST. Higher bandwidth allows more aggressive prefetching (up to 96 bytes), but even infinite bandwidth cannot justify the use of 512-byte loads, even though this load size produces the lowest read miss rate. The problem with large load sizes for `MMp3d` is that the cost of writes begins to dominate performance; writes account for as much as 45% of the MAST with a 512-byte load size.

`Blocked LU` exhibits the smallest improvement in read miss rates from sequential prefetching. The minimum miss rate produced by sequential prefetching is only 13%

lower than the minimum miss rate achieved without sequential prefetching. Most of the improvement comes from a reduction in false sharing misses.

Figure 5.16 shows that the MAST of `Blocked LU` is minimized with a load size of 64 bytes at all levels of bandwidth. Once again, writes account for a large portion of the MAST, especially at the larger load sizes. For example, with a 512-byte load size and low bandwidth, writes account for 30% of the stall time.

In summary, while aggressive sequential prefetching often improves the read miss rate of applications, it may not significantly reduce the stall time. In the particular case of programs with fine-grain sharing and lots of write operations (e.g., `MMp3d`), the minor improvements in the read miss penalty offered by aggressive prefetching may not compensate for a corresponding increase in the cost of writes.

5.3.3 Hybrid Prefetching

In this section, we investigate whether hybrid prefetching can improve on the performance of sequential prefetching. Hybrid prefetching has the potential to perform better, since it can prefetch with *any* fixed stride between blocks, while being selective about how aggressively to prefetch. In particular, the compiler may select aggressive prefetching for instructions dominated by cold misses, while using more conservative prefetching strategies for instructions that reference data that exhibit fine-grained sharing.

Our implementation of hybrid prefetching does not involve modifications to a real compiler. Instead, we collect the required stride information by profiling our programs during a simulation run. That is, we record a trace of the instructions with the highest miss rates and the addresses referenced by those instructions. For each instruction that generates a substantial number of misses, we process the trace to obtain the stride of access between every two consecutive references. We select the stride that occurs most frequently (and represents at least 25% of all references generated by the instruction) as the prefetching stride. Using this information, we manually instrument our programs with directives for modifying the instruction/stride table at run time. The number of blocks to prefetch is constant for all instructions (except in a few cases, where we limit prefetching to a single block due to fine-grain sharing), and we vary this number as one of the parameters for our study.

The performance of hybrid prefetching is dictated in large part by the access stride (in terms of cache blocks) and sharing patterns of applications. Both `Gauss` and `TGauss` exhibit unit stride, since most of the references in their inner loops are to consecutive cache blocks. Thus, both sequential and hybrid prefetching can be effective for `Gauss` and `TGauss`. In `MMp3d`, accesses to consecutive particles and adjacent (along the x axis) space cells result in references to alternating cache blocks, since these data structures are padded to fill two cache blocks. For `Blocked LU` the most common stride of access is 48 blocks, which is caused by accesses to columns in a matrix stored in row-major order. We would expect both `MMp3d` and `Blocked LU` to benefit from hybrid prefetching, since both of these programs have a significant fraction of references with a fixed, non-unit

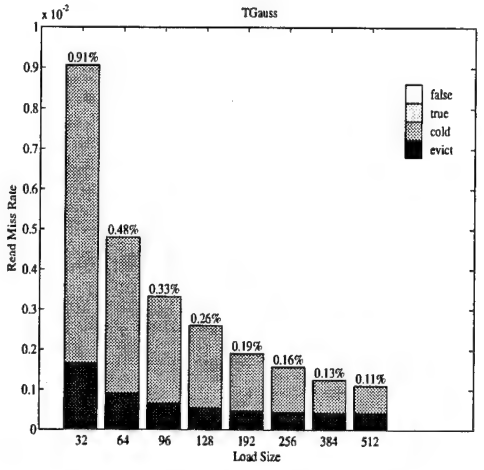


Figure 5.17: Read miss rate of TGauss under hybrid prefetching.

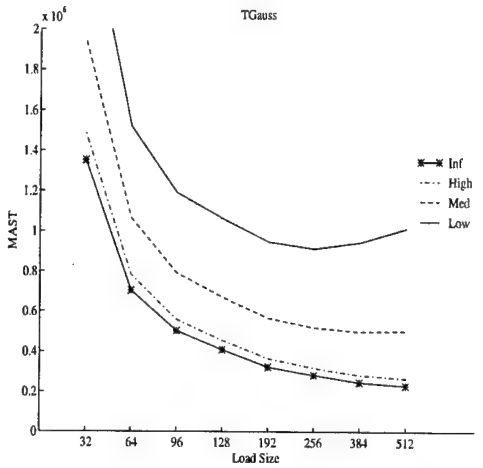


Figure 5.18: MAST of TGauss under hybrid prefetching.

stride. [3] For Barnes-Hut, the vast majority of accesses in the program have no regular stride, so this program is likely to pose serious problems for any prefetching strategy that depends on regular access patterns.

The sharing behavior of MMp3d and Blocked LU is similar in that both programs exhibit fine-grain sharing; the miss rate of both applications is dominated by true and false sharing misses. In the case of MMp3d, most of the misses (40%) are the result of true sharing of the data structure representing the wind tunnel space (the cell array). For Blocked LU, most of the misses are caused by false sharing of data in the main matrix. As a result, our implementation of hybrid prefetching only prefetches one additional block when satisfying a read miss to either of these data structures.

Figures 5.17-5.22 show the read miss rate and MAST of TGauss, MMp3d, and Blocked LU under hybrid prefetching. As expected, hybrid prefetching and sequential prefetching perform exactly the same for TGauss. For MMp3d hybrid prefetching produces slightly lower miss rates than sequential prefetching, and a much lower MAST with large load sizes. Hybrid prefetching offers the most improvement for Blocked LU, cutting the minimum read miss rate from 2.8% to 1%, and substantially reducing the minimum MAST achievable.

By comparing figures 5.14 and 5.20 we observe that the MAST incurred by hybrid prefetching is relatively insensitive to load size, while the stall time incurred by sequential prefetching increases dramatically with load size. Unlike hybrid prefetching, the stall time produced by aggressive sequential prefetching is heavily influenced by write buffer stalls. The main reason write stalls are kept under control in hybrid prefetching is that we use a conservative prefetching strategy on the cell array in MMp3d, while

[3] As seen in the previous section, sequential prefetching with a load size of 64 bytes is not particularly effective for MMp3d, but prefetching with a load size of 96 bytes is effective. The explanation for this behavior is the dominant number of references to alternating cache blocks in MMp3d. In this case, prefetching one block (a load size of 64 bytes) does not help, but prefetching two blocks does help.

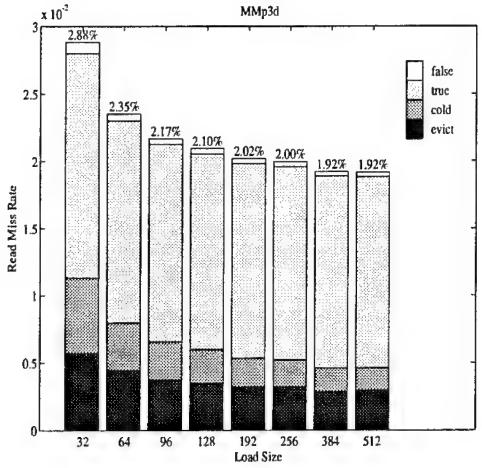


Figure 5.19: Read miss rate of `MMp3d` under hybrid prefetching.

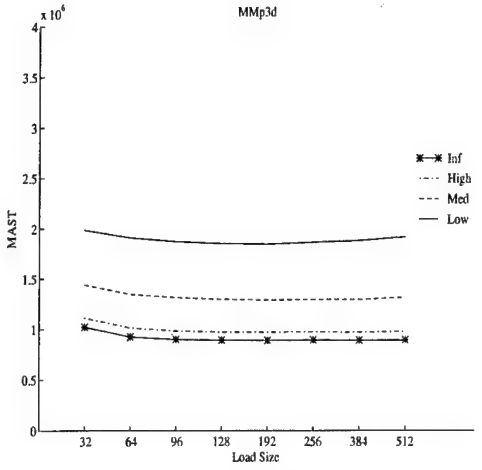


Figure 5.20: MAST of `MMp3d` under hybrid prefetching.

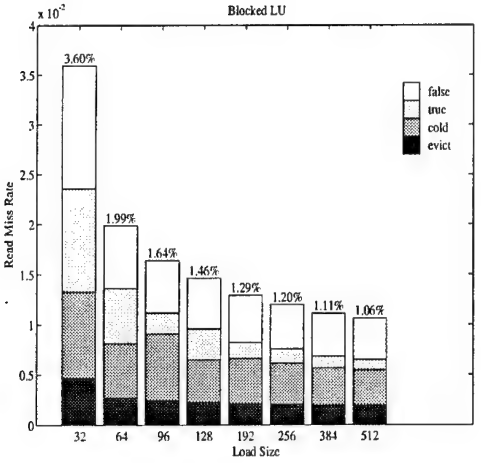


Figure 5.21: Read miss rate of `Blocked LU` under hybrid prefetching.

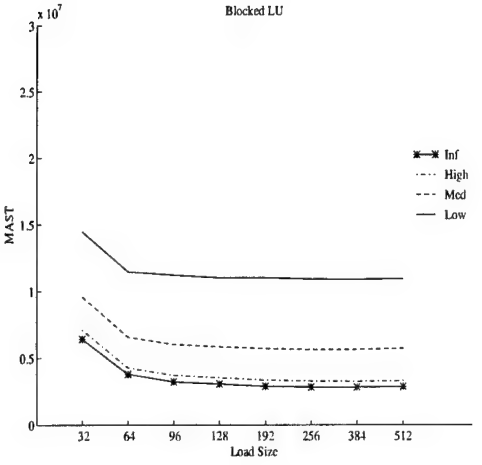


Figure 5.22: MAST of `Blocked LU` under hybrid prefetching.

simultaneously using an aggressive strategy on other data structures. Although our conservative strategy results in a slightly higher miss rate than would otherwise be possible with hybrid prefetching, it avoids the excessive stall time due to writes encountered under sequential prefetching.

By comparing figures 5.16 and 5.22 we can see that hybrid prefetching not only avoids excessive stall time due to writes, it also reduces the minimum MAST for each level of bandwidth. For `Blocked LU`, the minimum MAST produced by sequential prefetching under low bandwidth is 13.5M cycles, while the minimum MAST under hybrid prefetching is 10.9M cycles. Hybrid prefetching is even better under high bandwidth, decreasing the minimum MAST from 6.1M cycles to 3.2M cycles. In this case, the performance gap between hybrid and sequential prefetching increases with bandwidth, because higher bandwidth allows for more aggressive prefetching, which is beneficial (i.e., lowers the miss rate) in the case of hybrid prefetching, but not in the case of sequential prefetching (which can only benefit unit-stride accesses). Note that these improvements depend on a conservative implementation of hybrid prefetching for a select group of instructions; we only prefetch a single block on each read miss caused by any of five instructions, which together are responsible for 33% of the read misses in the program.

`Blocked LU` is an example of a program that exploits both of the properties that distinguish hybrid prefetching from sequential prefetching: a non-unit stride of access and a mixture of aggressive and conservative prefetching. The benefits of hybrid prefetching are limited however, because we avoid aggressive prefetching on five instructions that account for one third of the read misses. Perhaps by restructuring the program to reduce sharing, we could use aggressive prefetching on all the instructions, and reduce stall time even more.

To test this hypothesis, we modified `Blocked LU` to produce a new program called static blocked LU (`SBlocked LU`). In the modified program, each process works on a single set of data elements throughout its lifetime, rather than migrating among data elements in the interests of load balancing. Although `SBlocked LU` exhibits less sharing than `Blocked LU`, it does not keep all processors busy throughout the execution; on average, a processor drops out of the computation every three phases of the program.

Figure 5.23 shows the read miss rates of `SBlocked LU`. The minimum read miss rate produced by hybrid prefetching is 0.39%, which is a factor of 7 improvement over 32-byte blocks, and a factor of 5 improvement over the minimum read miss rate produced by sequential prefetching. Most of the improvements in the miss rate come from reductions in true and false sharing. As expected, `SBlocked LU` has lower miss rates than `Blocked LU`; the lowest miss rate of `SBlocked LU` is a factor of 2.7 smaller than the lowest miss rate of `Blocked LU`. More importantly, the improvement in the miss rate offered by prefetching is dramatically higher for `SBlocked LU`, a factor of 7 vs. a factor of 3.4 for `Blocked LU`.

Figure 5.24 shows the MAST of `SBlocked LU` as a function of the load size and available bandwidth. As seen in the figure, there is a significant drop in the cost of memory accesses as we increase the load size up to 96 bytes, beyond which the MAST performance improves very slowly. Nonetheless, hybrid prefetching performs much better than sequential prefetching, lowering the minimum MAST of sequential prefetching

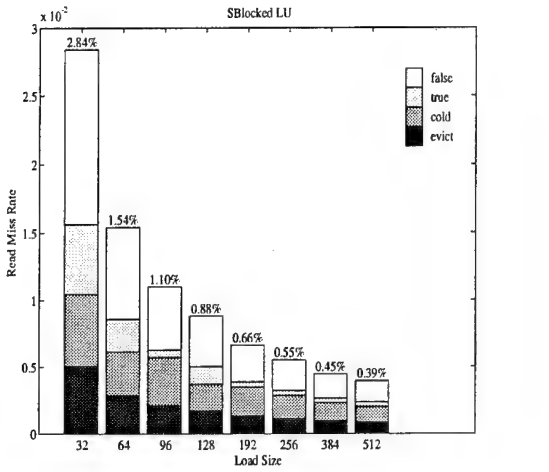


Figure 5.23: Read miss rate of `SBlocked LU` under hybrid prefetching.

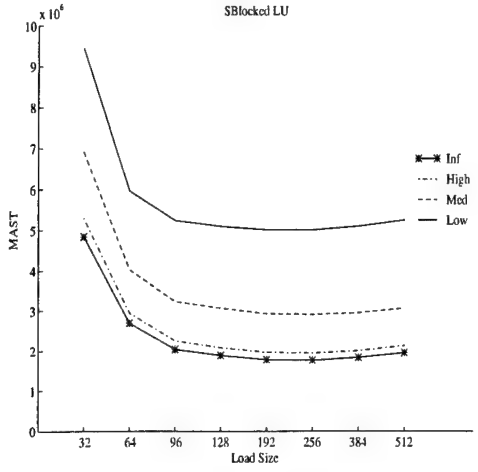


Figure 5.24: MAST of `SBlocked LU` under hybrid prefetching.

by 41% under low bandwidth, and 53% under high bandwidth. In contrast, hybrid prefetching lowered the minimum MAST of sequential prefetching for `Blocked LU` with low bandwidth by only 19%, and by 48% with high bandwidth. This example illustrates the enormous benefits of aggressive prefetching in the absence of fine-grain sharing.

In summary, hybrid prefetching is comparable to sequential prefetching for programs with unit-stride access (such as `TGauss`), but offers additional opportunities for prefetching for programs with large, regular stride accesses (such as `SBlocked LU`). By using a mixture of aggressive and conservative prefetching within a program, hybrid prefetching can offer the benefits of prefetching for instructions with a regular access pattern, while avoiding prefetching on instructions that result in excessive sharing (as in `MMp3d` and `Blocked LU`). Since hybrid prefetching need not use the same load size on every instruction, it is better able to translate an increase in bandwidth to an increase in load size. As a result, the benefits of hybrid prefetching relative to sequential prefetching tend to increase with bandwidth.

5.3.4 Comparison of Prefetching Techniques

In this section we evaluate the success of cache-miss-initiated prefetching by examining the overall effect on running time of each technique. We also compare cache-miss-initiated prefetching with software prefetching [Callahan *et al.*, 1991; Mowry *et al.*, 1992], which does not require misses to initiate prefetching.

As with hybrid prefetching, we implemented software prefetching by hand. We use the miss rate information gathered for hybrid prefetching to determine the instructions that can benefit from prefetching. After identifying the most important instructions, we manually inserted prefetches so that data blocks are received just before they are required. In order to hide the latency of prefetching without generating substantial instruction execution overhead, we perform loop unrolling and splitting wherever nec-

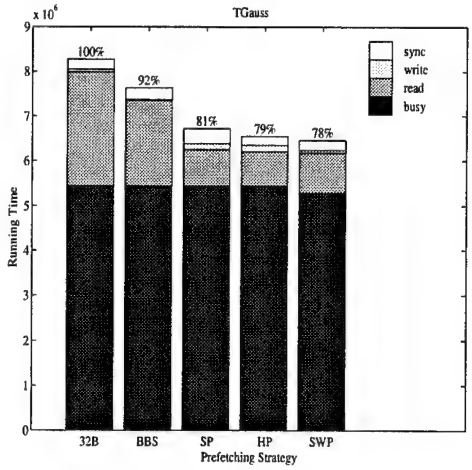


Figure 5.25: Running time of TGauss with low bandwidth.

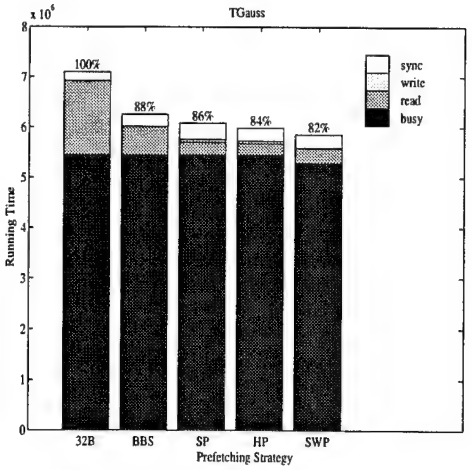


Figure 5.26: Running time of TGauss with high bandwidth.

essary. Each prefetch instruction prefetches a single cache block in read mode; that is, we do not implement block and exclusive prefetches.

Figures 5.25-5.32 present a comparison of the running time produced by each of the techniques under low and high bandwidth assumptions for each of our applications. In these figures each column represents a different prefetching technique: 32-byte blocks with no prefetching (32B), the best block size for a given program (i.e., the block size that produces the smallest running time) with no prefetching (BBS), sequential prefetching with the load size that produces the smallest running time (SP), hybrid prefetching with the load size that produces the smallest running time (HP), and software prefetching (SWP). The number on the top of each column is the running time of that technique as a percentage of the running time for the base case of 32-byte blocks with no prefetching. Within a column running time is broken into busy time, read stall time, write stall time, and synchronization overhead.

Figure 5.25 shows the running time of TGauss with low bandwidth. As seen in the figure, sequential, hybrid, and software prefetching perform roughly the same for this program, improving its running time by about 20%. Using the best block size for this program (64 bytes) and no prefetching improves performance by only 8%.

It is interesting to note that software prefetching has a slightly lower busy time than the other techniques, despite the need for prefetching instructions. The reason for this counter-intuitive behavior is that in our implementation of software prefetching we unrolled the main computational loop in TGauss several iterations more than a standard compiler would have done. Without this aggressive unrolling, software prefetching introduces high instruction overhead.

In terms of the read stall overhead, software prefetching performed slightly worse than sequential and hybrid prefetching since, at this level of bandwidth, more than 30% of the prefetches issued under software prefetching are completed too late to avoid a miss.

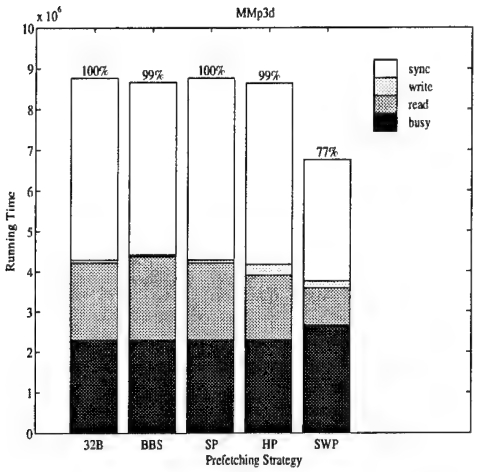


Figure 5.27: Running time of `MMp3d` with low bandwidth.

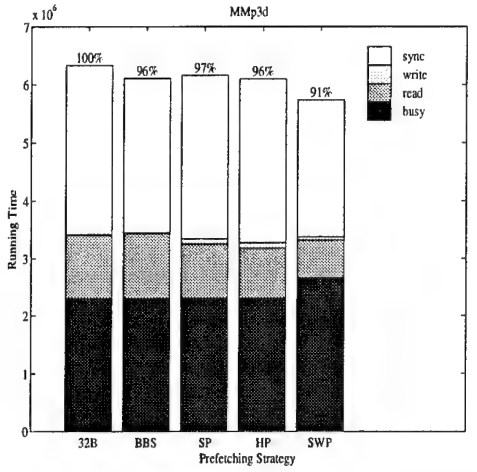


Figure 5.28: Running time of `MMp3d` with high bandwidth.

Figure 5.26 shows running times for `TGauss` under high bandwidth. At this level of bandwidth, the best block size (128 bytes) performed almost as well as the other techniques. Software prefetching performs better in terms of read stall time under high bandwidth, where only 9% of the prefetches are received late.

Figures 5.27 and 5.28 present running times for `MMp3d` under low and high bandwidth, respectively. In both figures we see that the cache-miss-initiated prefetching techniques are not successful at reducing the execution time significantly in comparison to the base architecture. Software prefetching, on the other hand, substantially improves running time (especially under low bandwidth), even though it increases the busy time.

As seen in figures 5.29 and 5.30, both hybrid prefetching and software prefetching are able to improve the running time of `Blocked LU` in comparison to the other techniques. Software prefetching performs better regardless of bandwidth because our implementation of hybrid prefetching is conservative on certain instructions. This conservative approach produces a higher read miss rate for hybrid prefetching (2.1% vs 1.6% at low bandwidth, and 1.5% vs 0.7% at high bandwidth).

Finally, figures 5.31 and 5.32 show the running time of `SBlocked LU` for each of the techniques. As with `Blocked LU`, both hybrid and software prefetching perform much better than the other techniques. Since `SBlocked LU` admits more aggressive cache-miss-initiated prefetching, hybrid and software prefetching offer comparable performance. Large block sizes and sequential prefetching produce very limited performance improvements, whereas hybrid and software prefetching improve the execution time by 20% to 24%, depending on bandwidth.

These results illustrate the circumstances under which cache-miss-initiated prefetching is most effective. If programs exhibit a regular access pattern, then each miss can prefetch a lot of data, and avoid future misses. Among the cache-miss-initiated techniques, only hybrid prefetching can adapt to large stride access patterns, and can tailor the amount of data prefetched on a miss according to the sharing behavior of each in-

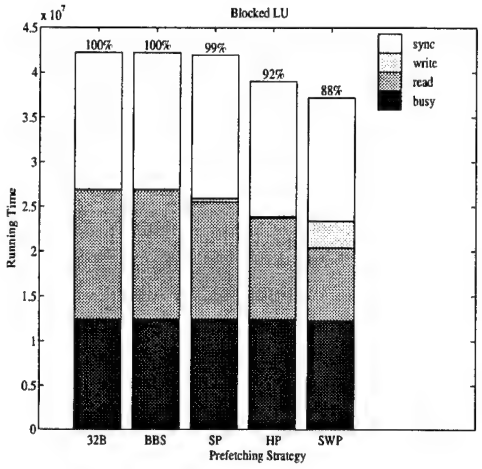


Figure 5.29: Running time of `Blocked LU` with low bandwidth.

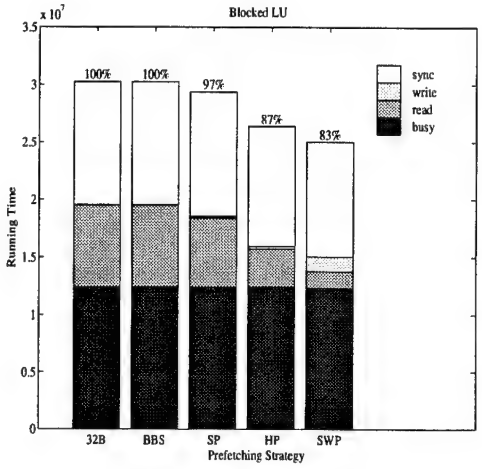


Figure 5.30: Running time of `Blocked LU` with high bandwidth.

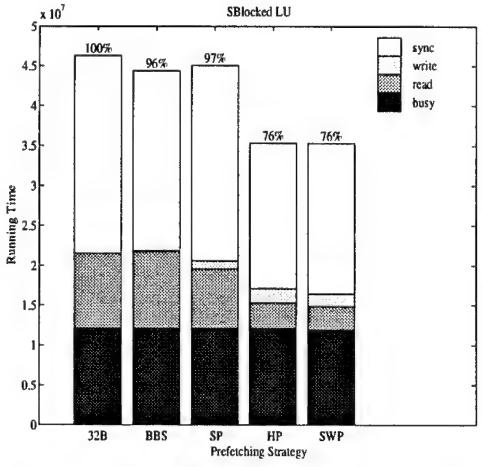


Figure 5.31: Running time comparison of `SBlocked LU` under low bandwidth.

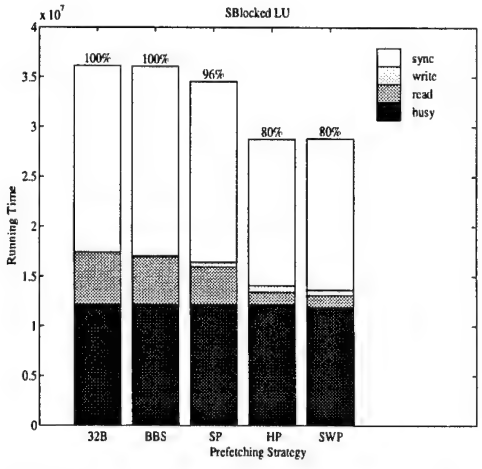


Figure 5.32: Running time comparison of `SBlocked LU` under high bandwidth.

struction. In the best case, hybrid prefetching offers performance comparable to software prefetching, which need not wait for a miss before issuing prefetches.

5.4 Summary

In this chapter we used execution-driven simulation of parallel programs on a scalable cache-coherent machine to study the performance of three cache-miss-initiated prefetching techniques: large cache blocks, sequential prefetching, and hybrid prefetching. Large cache blocks and sequential prefetching are well-known prefetching strategies. Hybrid prefetching is a novel technique combining hardware and software support for stride-directed prefetching.

Our simulation results showed that large cache blocks rarely provide significant performance improvements; the incremental improvement in the miss rate gained by using larger blocks is simply too small to offset a corresponding increase in the miss penalty. Our results also showed that sequential prefetching improves on the performance of large cache blocks by alleviating false sharing and high miss penalties. A comparison of sequential and hybrid prefetching shows that the latter technique performs at least as well as the former, as it can prefetch with large strides between blocks, while being selective about how aggressively to do so. In fact, given sufficiently high bandwidth and regular memory addressing, hybrid prefetching can perform as well as software prefetching. We conclude that among the cache-miss-initiated prefetching techniques we consider, hybrid prefetching is the only technique that offers significant performance improvements for scalable multiprocessors.

6 Tolerating Remote Access Overhead with Update-Based Coherence Protocols

The two previous chapters considered whether aggressive prefetching could significantly improve the performance of parallel applications in the presence of high network and memory bandwidth. In the same vein, this chapter considers whether high remote access bandwidth can significantly affect the tradeoff between update-based coherence protocols and their invalidate-based counterparts on scalable multiprocessors.

The cache coherence protocol determines how data is moved among the caches in the machine, ensuring that data is frequently found in the local cache, while preventing processors from using stale data. Under a write-update (WU) protocol [McCreight, 1984; Thacker and Stewart, 1987; Thacker *et al.*, 1992], each time a processor writes shared data, the coherence protocol broadcasts the new value to every other processor caching that data. Under a write-invalidate (WI) protocol [Goodman, 1983; Papamarcos and Patel, 1984; Lenoski *et al.*, 1990], a write to a shared cache block causes the coherence protocol to mark as invalid all other cached copies of the block.

The advantage of WU is that each processor receives and stores the update as it occurs, thus preventing future cache misses when the new value is needed. This property is particularly helpful when many processors read the updated values between successive write operations to the data [Eggers and Katz, 1988]. The disadvantage of WU is that every write operation to shared data requires that updates be sent over the network, even if no processor accesses the data between successive writes. WI achieves superior performance when cache blocks are written many times by a single processor before being accessed by any other processor [Eggers and Katz, 1988]. In most cases, WI results in higher miss rates, but fewer communication operations.

Previous studies comparing WI and WU protocols on bus-based machines have offered mixed results [Archibald and Baer, 1986; Eggers and Katz, 1988; Veenstra and Fowler, 1994b]. In general, the comparison depends on the relative cost of reads and writes, and the sharing patterns exhibited by programs. WI protocols are currently used in the vast majority of hardware-coherent systems however. Although the broadcast nature of a shared bus allows the coherence protocol to update many processors with a single transaction, WI still performs better than WU in most cases because (1) the communication bandwidth available per processor on these machines is usually very limited and is rapidly consumed by the excessive number of transactions produced by

WU and (2) the cost of an update transaction on the bus is roughly the same as the cost of rereading an invalidated cache block, and there are likely to be many more transactions under WU than under WI.

Scalable, network-based machines offer a very different environment for comparing WU and WI. These machines may incorporate relaxed consistency and write buffers (which reduce the cost of writes), or may use page-based coherence [Bisiani and Ravishankar, 1990; Carter *et al.*, 1991; Wilson and LaRowe, 1992] (which results in high latency and bandwidth requirements for page faults).

Previous studies of scalable cache-coherent architectures [Dahlgren *et al.*, 1994; Dahlgren and Stenstrom, 1994] have shown that update-based protocols, with some additional sophisticated hardware to reduce the amount of network traffic, can outperform their invalidate-based counterparts. However, future scalable cache-coherent machines will likely have very high communication bandwidth and remote access latency. Under these architectural assumptions, one might expect a pure WU protocol to perform as well as more hardware-intensive (and therefore costly) implementations of the protocol, since the extra traffic associated with pure WU would not have a significant performance impact.

Our evaluation of pure WU and WI protocols as a function of bandwidth and block size does not confirm this expectation, however. Detailed simulations of these two classes of protocol show that the excessive network traffic caused by update protocols significantly degrades performance, even with infinite bandwidth. We trace the poor performance of WU to a variety of factors that are independent of bandwidth, but are all related to an excessive number of updates. Motivated by this observation, we categorize the coherence traffic generated by a WU protocol to quantify the amount of update traffic necessary for correct execution. The results of this analysis show that, for most applications, more than 90% of all updates are useless. Our analysis also pinpoints the application characteristics responsible for useless update traffic.

The dominance of useless traffic in our experiments led us to consider the extent to which techniques for improving the performance of WU protocols reduce useless traffic. We study several software and hardware techniques, such as hybrid invalidate/update protocols, a data re-mapping strategy, and coalescing write buffers. Although all the techniques we consider significantly reduce the useless traffic associated with update-based protocols, there is still the potential for further reductions in traffic. We suggest several directions for further eliminating useless traffic in update-based protocols.

Our work differs from most previous studies on coherence protocols in that our main goal is not to determine whether to use WI or WU for a particular architecture; we use WI running time results simply as a basis for comparison. Rather, in the same vein as [Dubois *et al.*, 1993], which examined useless misses in WI protocols, our main goals are to categorize updates in terms of their usefulness, and to compare techniques intended to improve the performance of WU protocols with respect to our categorization.

Our work is similar to recent work by Dahlgren *et al* [Dahlgren *et al.*, 1994; Dahlgren and Stenstrom, 1994] in that we consider some of the same techniques for optimizing WU protocols. In that work, write caches and dynamic hybrid protocols achieved better

performance than WI. Our work is distinguished by our focus on the source and usefulness of updates (and their associated acknowledgements), which allows us to relate program modifications to protocol improvements, evaluate protocol optimizations with respect to reductions in network traffic, and suggest new optimizations to update-based protocols. In addition, our evaluation of the running times associated with optimizations to WU protocols is intended to capture the impact of bandwidth and block size on performance, which is not the focus of previous work. Furthermore, by treating both bandwidth and block size as parameters, and by considering several additional modifications to a WU protocol, we explore interactions between these factors not previously observed.

The remainder of this chapter is organized as follows. We first describe our simulation methodology, performance metrics, and application workload in section 6.1. Section 6.2 presents the miss rate and network traffic associated with each of our applications under the non-optimized WI and WU coherence protocols. In section 6.3, we explore the effects of bandwidth and cache block size on the performance of the two protocols for each of our programs, using running time as our main evaluation metric. In section 6.4, we explain the causes of poor WU performance, introduce our categorization of update traffic, and evaluate several optimizations to WU with respect to the categorization and execution time. Section 6.5 investigates the performance of the protocols we consider on next-generation architectures. Section 6.6 contains a summary of our findings.

6.1 Methodology and Workload

6.1.1 Multiprocessor Simulation

We use detailed execution-driven simulation to model a scalable multiprocessor with 32 nodes. Once again, each node in the simulated machine contains a single processor, a write buffer, cache memory, local memory, directory memory, and a network interface. Each processor has a 64 KB direct-mapped data cache. The data cache block size, the unit of fetching and coherence, is one of the parameters in our simulations; we consider small (16 bytes), medium (64 bytes), and large block sizes (256 bytes).

All instructions are assumed to take one cycle. A data read that hits in the cache also takes 1 cycle. Read misses stall the processor until the read request is satisfied. Writes go into an 8-entry write buffer and take 1 cycle, unless the write buffer is full, in which case the processor stalls until an entry becomes free. Reads are allowed to bypass writes that are queued in the write buffers. Furthermore, reads have priority over writes for accessing the cache and memory bus, but are prevented from accessing the cache while a write is updating it.

In order to model the limited number of pins in real processors, reads and writes contend for off-chip access. The cache is assumed lockup-free, so that the cache can be accessed while off-chip references are pending. Reads or writes may be locked out of the cache whenever the cache is being invalidated or updated from the outside. The lockout period is 2 cycles and occurs (if necessary) at the end of a memory bus operation.

A pipelined memory bus (clocked at one fourth of the speed of the processor) connects the main components of each machine node. A new bus operation can start every 20 processor cycles. The memory can provide the first word 32 processor cycles after the request is issued. The width of the memory bus (memory bandwidth) is another parameter of our study and varies according to the network path width.

As in previous chapters, the simulator implements a full-map directory for controlling the state of each block of memory. Shared data are interleaved across the machine at the block level. Each node contains the directory for the memory associated with that node.

The characteristics of the interconnection network we use in this chapter are exactly the same as described in chapter 5. We experiment with two finite bandwidth networks with 16 and 64-bit wide data paths. In all our machine configurations we assume that the bus width is the same as the network link width.

Our WI protocol keeps caches coherent using the DASH protocol with release consistency [Lenoski *et al.*, 1990]. In our WU implementation, a processor writes through its cache to the home node. The home node sends updates to the other processors sharing the cache block, and a message to the writing processor containing the number of acknowledgements to expect. Sharing processors update their caches and send an acknowledgement to the writing processor. Since we assume release consistency, the writing processor does not have to wait for update acknowledgements before continuing execution; the processor only stalls waiting for acknowledgements at a lock release point. Under this protocol, blocks are evicted from the cache only due to replacement.

Our pure WU implementation actually includes two optimizations. First, when the home node receives an update for a block that is only cached by the updating processor, the acknowledgement of the update instructs the processor to retain future updates since the data is effectively private. This optimization is analogous to the shared line on the bus of the Dragon [McCreight, 1984] and Firefly [Thacker and Stewart, 1987] multiprocessors. Second, when a parallel process is created by *fork*, we flush the cache of the parent's processor, which eliminates useless updates of data initialized by the parent but not subsequently needed by that processor.

6.1.2 Performance Metrics

The focus of this chapter is on a categorization of the coherence traffic in update-based protocols (refer to appendix A). However, we also present the running time of the parallel section of the code and its components (busy time, read latency, write latency, and blocked time). In addition, we consider read miss rates and the total network traffic. We concentrate on read misses because we assume relatively deep write buffers and release consistency, which serve to hide the cost of writes. The read miss rate is computed solely with respect to shared references; that is, the read miss rate is defined as the total number of read misses on shared data divided by the total number of reads to shared data. We classify cache misses using the algorithm described in [Dubois *et al.*, 1993] as extended in appendix A.

Application	Shared Refs	Shared Reads (% of shared refs)	Shared Writes (% of shared refs)
Mp3d	5.1 M	60 %	40 %
Barnes-Hut	19.0 M	97 %	3 %
CG	6.8 M	98 %	2 %
Em3d	6.7 M	91 %	9 %
Blocked LU	47.3 M	90 %	10 %
SOR	20.7 M	85 %	15 %

Table 6.1: Memory reference characteristics on 32 processors.

6.1.3 Workload

Our application workload consists of six parallel programs, some of which have been described in previous chapters: Mp3d, Barnes-Hut, CG, Em3d, Blocked LU, and SOR. Mp3d simulates 30000 particles for 5 steps. Barnes-Hut simulates 2K bodies for 4 time steps. CG uses the conjugate gradient method to compute an approximation to the smallest eigenvalue of a 1400 $\times$ 1400, sparse, symmetric positive definite matrix with 78148 non-zero elements. Our CG implementation is a C version of the CG kernel from the NAS parallel benchmarks suite [Bailey et al., 1994]. Em3d simulates electromagnetic wave propagation through 3D objects. We simulate 20000 electric and magnetic nodes connected randomly, with a 10% probability that neighboring nodes reside in different processors. We simulate the interactions between nodes for 30 iterations. Blocked LU and SOR run on 384 $\times$ 384 matrices.

The code for our applications was generated by a MIPS C compiler with -O2 optimization flag. Table 6.1 summarizes the distribution of shared references in our applications.

6.2 Miss Rates and Message Traffic of WI vs. WU

In this section we examine the miss rates and network traffic produced by WU and WI protocols on our application suite, so as to quantify the lower miss rates of WU and reduced message traffic of WI.

Figure 6.1 presents the read miss rate of our applications for WI (left) and WU (right) (assuming 64-byte cache blocks). The percentage at the top of each column represents the percent of all read references to shared data that result in a miss; within a column misses are classified as either eviction, cold start, true sharing, or false sharing misses. Since WU only removes blocks from the cache due to evictions, WU has only eviction and cold-start misses; sharing-related misses are eliminated since multiple writes to the same block can be issued by one or more processors without causing any of them to lose that block.

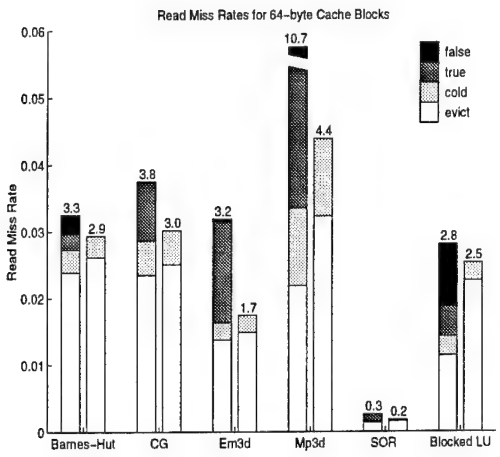


Figure 6.1: Miss rate under WI (left) vs. WU (right) for 64-byte cache blocks.

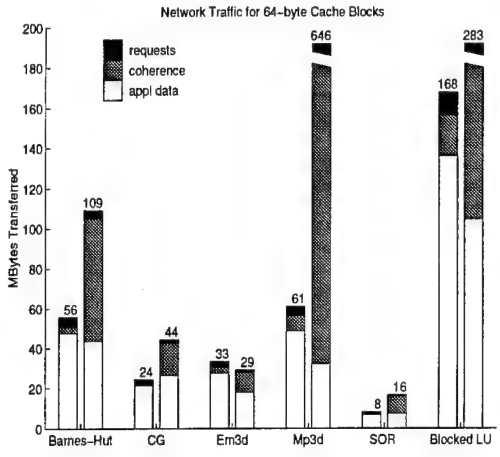


Figure 6.2: Bytes transferred under WI (left) vs. WU (right) for 64-byte cache blocks.

From figure 6.1 we can see that, as expected, WU always results in lower read miss rates, even though WI usually exhibits lower replacement miss rates as invalidations effectively free up cache space. For `Mp3d`, `Em3d`, and `SOR` the difference in read miss rate is particularly large; the read miss rate under WI is a factor of 50-140% higher than under WU. For other applications, this difference is not as significant, as replacements dominate the miss rate under both protocols. These general effects are consistent across the block sizes we consider, although the magnitude of the miss rate differences varies slightly for other block sizes.

Although WU produces lower miss rates, it is at the cost of many update messages. Figure 6.2 presents the total number of bytes transferred by each coherence protocol for each application (again assuming 64-byte cache blocks). The number at the top of each column represents the number of MBytes transferred by the two protocols; within a column the traffic is classified as either data, coherence (includes invalidations, updates, and acknowledgements), and requests. This figure clearly shows that in terms of the number of bytes transferred, WU requires much more network traffic than WI, except in the case of `Em3d`. For `Mp3d` there is more than an order of magnitude difference in the amount of data transferred by the network, 646 Mbytes for WU vs. 61 Mbytes for WI. In terms of the number of messages sent, this corresponds to 70 M messages for WU vs. 2.5 M messages for WI. `Barnes-Hut` and `SOR` have lower miss rates, and therefore require less communication, but the difference between WU and WI is again substantial; WU transfers nearly twice as many bytes as WI, and requires about 5 times as many messages. Again, this effect is consistent across block sizes.

`Em3d` is an exception as the network transfers less data under WU than under WI, independently of the cache block size. The most important reason for this effect is that the increase in coherence traffic associated with WU is somewhat limited, since cache blocks effectively shared are very infrequently written in this program.

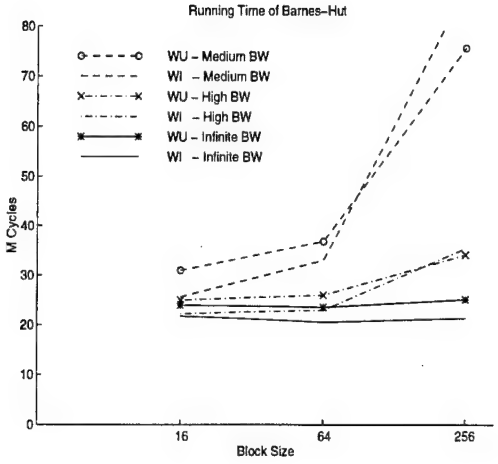


Figure 6.3: Running time of `Barnes-Hut`.

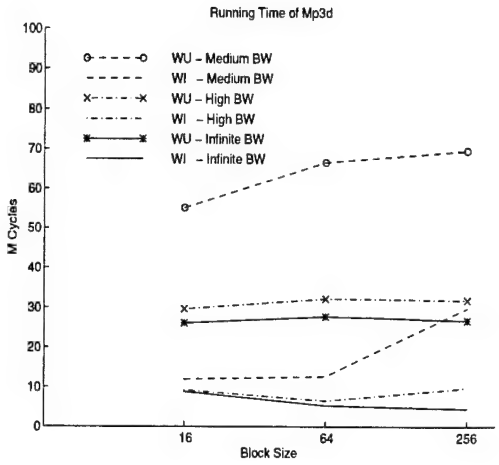


Figure 6.4: Running time of `Mp3d`.

Comparing the amount of data and coherence traffic involved in our applications (shown in figure 6.2), we observe that the network traffic associated with WU is dominated by updates and their corresponding acknowledgements (rather than misses) in most cases, and this traffic usually grows significantly with an increase in block size. Even if a larger block size produces a lower miss rate, and hence less network traffic due to misses, the reduction in network traffic due to fewer misses is overwhelmed by an increase in the number of updates.

In summary, WU can lower the miss rate by roughly 10-60% over WI, while increasing network traffic by as much as an order of magnitude. The benefits of the lower miss rate depend on the remote access latency of the machine, while the costs of the additional network traffic depend on the available network bandwidth. In the next section, we consider whether future increases in network bandwidth will be sufficient to resolve this tradeoff in favor of WU.

6.3 The Effect of Bandwidth and Block Size on Performance

In this section we consider whether expected increases in network and memory bandwidth enable a pure WU protocol to outperform WI on a scalable machine. We also investigate how changes in block size affect our comparison of the protocols.

Figures 6.3-6.8 present the running time of each application in our suite under the two different protocols, for three levels of bandwidth and a range of cache block sizes. As seen in the figures, WI performs better than WU for `Barnes-Hut` and `Mp3d`; the two protocols achieve comparable performance for `CG` and `SOR`; and WU outperforms WI for `Em3d` and `Blocked LU`.

In general, any comparison between WI and WU protocols must consider the neg-

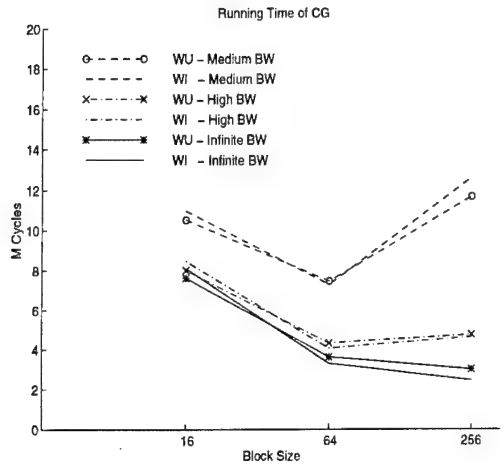


Figure 6.5: Running time of CG.

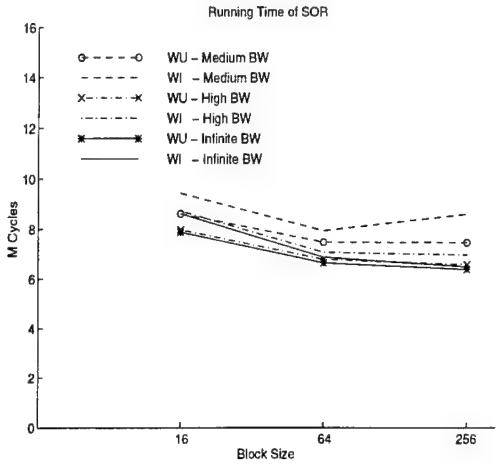


Figure 6.6: Running time of SOR.

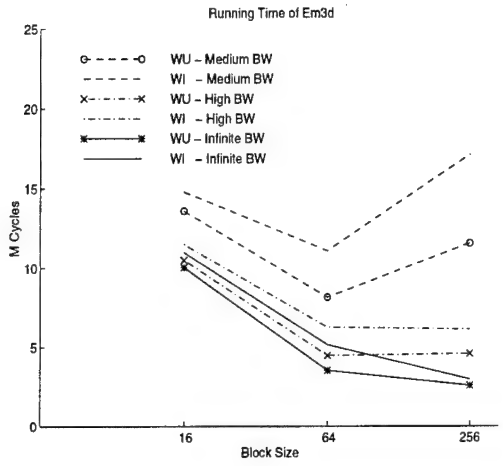


Figure 6.7: Running time of Em3d.

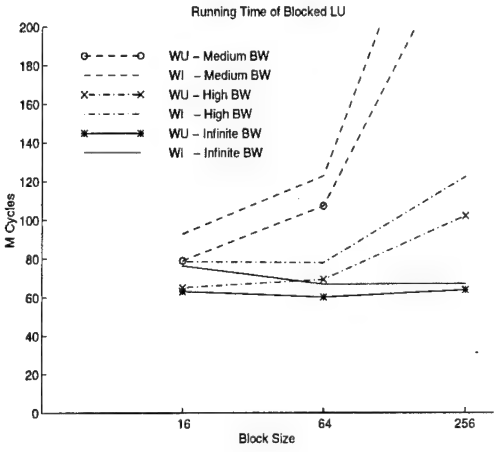


Figure 6.8: Running time of Blocked LU.

ative impact of the higher cost of read accesses under a WI protocol, in contrast with the potential degradation caused by the excessive network traffic generated under a WU protocol. Figures 6.3-6.8 illustrate the effect of bandwidth and cache block size on this tradeoff. Medium bandwidth significantly degrades the performance of 256-byte blocks (which produce the lowest miss rates for all applications and protocols, except Barnes-Hut) regardless of the protocol. Increasing the bandwidth alleviates the performance degradation associated with the larger blocks, while not significantly affecting the performance of the smaller blocks in most cases.

High bandwidth diminishes the impact of the update traffic independently of the block size and could potentially provide an advantage to the WU protocol. Our simulation results do not indicate a performance advantage of WU over WI, however. Given infinite bandwidth, WU performs slightly better (if at all) than WI for small cache

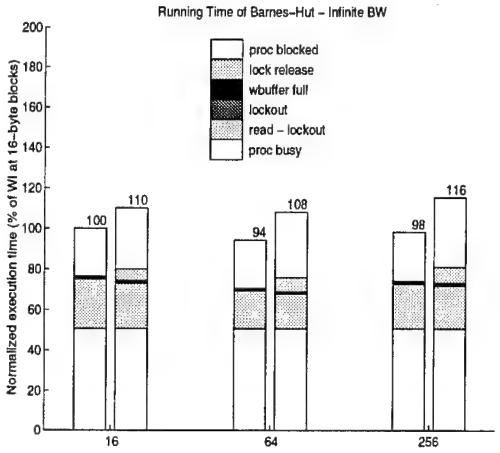


Figure 6.9: Running time of `Barnes-Hut` under WI (left) vs. WU (right).

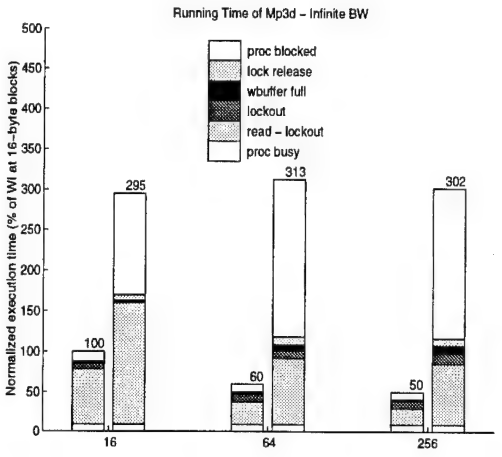


Figure 6.10: Running time of `Mp3d` under WI (left) vs. WU (right).

blocks, while the performance of the two protocols with large blocks is indistinguishable for three of our applications. The exception is `Mp3d`, for which WU is significantly worse across all block sizes, even though WU results in lower miss rates. The fact that infinitely-wide data paths do not enable WU to perform much better than WI suggests that bandwidth alone is not enough to justify using WU.

The enormous network traffic associated with WU causes several forms of performance degradation ultimately due to an increase in network and memory congestion. The degradation may manifest itself in many different ways, such as (1) increased read latency, (2) increased write latency induced by processor stalls due to full write buffers, (3) increased synchronization overhead whenever processors must wait for update acknowledgements before releasing locks, and (4) increased synchronization latencies whenever the processor holding a lock has its (blocking) reads delayed by previous messages. In addition, an excessive number of update messages may cause processors to be locked out of their caches frequently.

We can observe the contribution to running time of these effects in figures 6.9-6.14. These figures break down the cumulative running time of processors (normalized to the time for WI with 16-byte blocks) under infinite bandwidth. The WI performance is shown on the left of each pair of bars; WU is on the right. The categories of time are (from top to bottom): processor blocked (or, in other words, lock acquire) overhead, lock release (including write buffer flush) overhead, stall time due to a full write buffer, stall time when a processor cannot access its cache (lockout), read access cost minus the lockout time, and processor busy time.

These figures show that each effect described above contributes to the tradeoff between WU and WI. `Barnes-Hut` (figure 6.9) performs better under WI than WU due to the relatively high (and constant) cost of read accesses, and the extra lock release and processor blocked overheads under WU. For `Mp3d`, read latencies and processor blocked

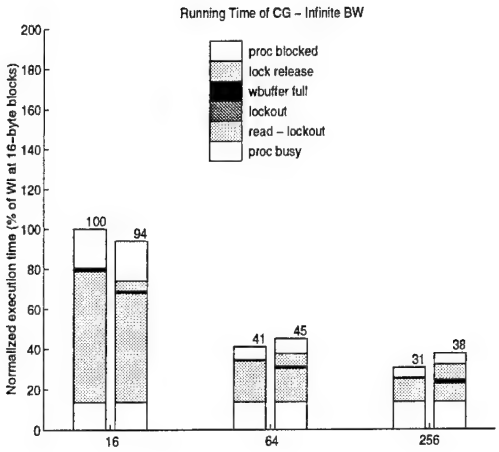


Figure 6.11: Running time of `CG` under WI (left) vs. WU (right).

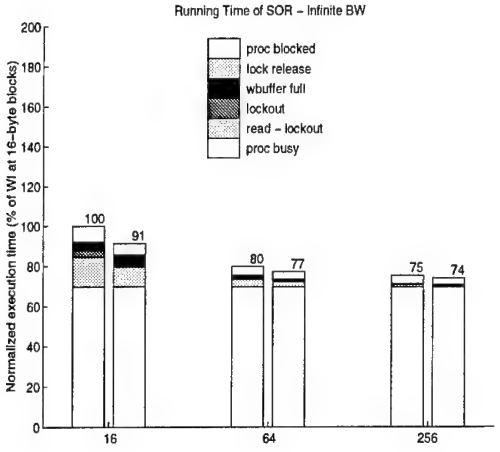


Figure 6.12: Running time of `SOR` under WI (left) vs. WU (right).

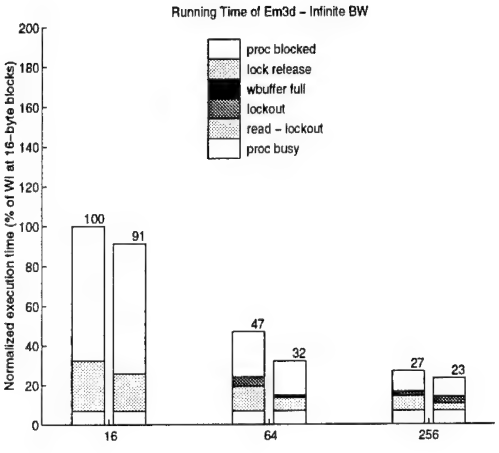


Figure 6.13: Running time of `Em3d` under WI (left) vs. WU (right).

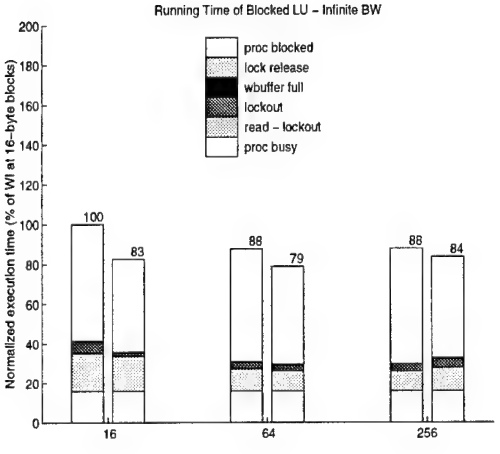


Figure 6.14: Running time of `Blocked LU` under WI (left) vs. WU (right).

overheads are extremely high for WU for all block sizes, and dominate the comparison against WI. Lock release overheads hurt WU performance for `CG` (figure 6.11), especially with the larger block sizes. The performance of the two protocols is comparable in most cases for `SOR`. For `Em3d`, WU performs better than WI, especially with 64-byte blocks; the performance of WU with 256-byte blocks is degraded by lockout overhead. For `Blocked LU`, the read latency under WU is slightly lower than under WI for 16 and 64-byte blocks. With 256-byte blocks the latency of reads under WU is actually larger than under WI, even though the WU miss rate is lower. The performance advantage obtained by WU is due to significantly lower processor blocked overheads than under WI. The reason for this effect is that `Blocked LU` is plagued by a large sequential processing component, which is reduced as a result of the fewer misses under a WU protocol.

It is clear from these figures that a lack of bandwidth is not the only problem with WU. Figures 6.3-6.6 showed that no amount of bandwidth enables WU to perform significantly better than WI in all cases. Although hardware techniques designed to alleviate a particular source of overhead, such as cache lockout, might help, the most significant source of overhead is the update messages themselves. We will now consider how to reduce the network traffic associated with WU.

6.4 Improving Write Update Performance

The previous section showed that high network and memory bandwidth is not enough to enable WU to consistently outperform WI. The excessive number of updates produced by WU introduces several performance problems, all of which could be alleviated by reducing the number of messages used by the WU protocol. We first consider how many updates are actually required for correct execution of the program, and then evaluate techniques for combining multiple updates in one message and eliminating useless updates.

6.4.1 Useful vs. Useless Updates

In order to investigate ways in which to eliminate update messages, we classify updates in terms of their usefulness to the processors receiving them. We classify updates as *useful* and *useless*. An update is useful if the processor references the updated word before the next update of that word arrives; otherwise the update is useless. Intuitively, useful updates are those updates required for correct execution of the program, while useless updates could be eliminated entirely and not affect the correctness of the execution.

We divide useless updates into four categories: *proliferation*, *false*, *replacement*, and *termination* updates. Update messages are classified at the end of an update's lifetime, which happens when it is overwritten by another update to the same word, when the cache block containing the updated word is replaced, or when the program ends. If, between two updates to a word, the cache block containing the word is not referenced, then we classify the first update as a proliferation update. If another word in the same block is referenced, then we classify the first update as a false (sharing) update. If

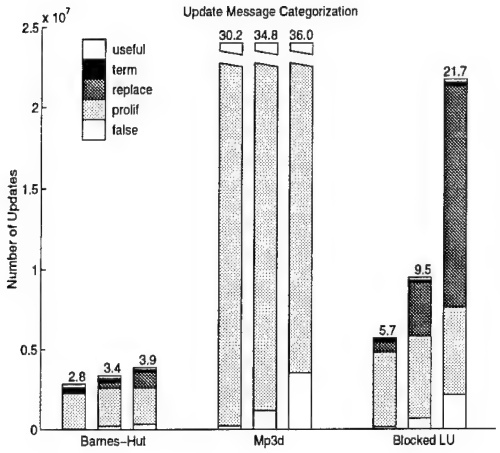


Figure 6.15: Categorization of updates for group 1 for blocks of 16 (left), 64 (center), and 256 (right) bytes.

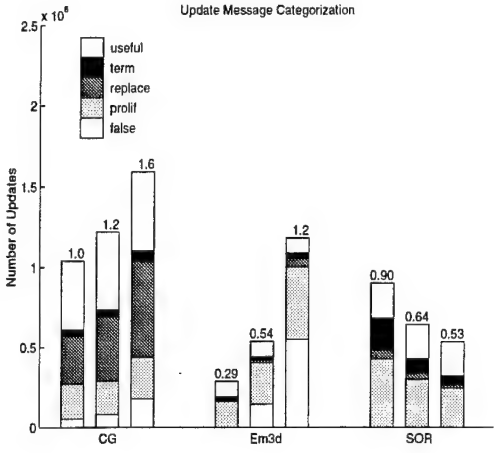


Figure 6.16: Categorization of updates for group 2 for blocks of 16 (left), 64 (center), and 256 (right) bytes.

a processor receives an update to a word and, before referencing the word, the corresponding cache block is replaced, then we classify the update as a replacement update. If a processor receives an update and the program terminates without referencing the block again, then we classify the update as a termination update.

This categorization is fairly straightforward, except for our false update class. Successive (useless) updates to the same word in a block are classified as proliferation instead of false sharing updates, if the receiving processor is not concurrently accessing other words in the block. Thus, our algorithm classifies useless updates as proliferation updates, unless *active* false sharing is detected or the application terminates execution.

Although by no means unique, our categorization is intuitive, while being sufficiently simple to compute, as it does not require any future knowledge of sharing behavior or an excessively large amount of memory. Greater details about the categorization and the algorithms we use in our simulations can be found in appendix A.

An analysis of the update messages sent during our simulation experiments shows that the number of useless updates is extremely high: more than 95% of all updates sent during execution of `Barnes-Hut`, `Mp3d`, and `Blocked LU` are useless, while between 60% and 90% of all updates in `CG`, `Em3d`, and `SOR` are useless. Despite the two optimizations described in section 6.1, which eliminate useless updates for data that is effectively private and for data that is initialized by a parent process prior to a *fork*, the vast majority of the remaining updates are still useless.

Figures 6.15 and 6.16 present the number and types of useless updates found in our programs. We separate the applications into two groups according to the percentage of useless updates found in the applications. Applications with more than 90% useless updates consistently across block sizes are in the first group. The other applications are placed in the second group. The number on top of each column represents the total number of updates (in millions).

Proliferation updates clearly dominate in two of the programs, Barnes-Hut and Mp3d. Proliferation updates are also an important factor for Blocked LU, but replacement updates dominate when using the largest block size we consider. In the case of Em3d, both proliferation and false updates are significant for 256-byte cache blocks. Useful and replacement updates are major contributors in CG, while useful and proliferation updates are significant in SOR.

In most cases, increasing the block size leads to more widespread sharing of cache blocks, which in turn causes an increase in the number of useless updates. The only exception to this trend is SOR, which exhibits a slight reduction in the number of updates as we increase the block size. This result is mainly caused by differences in sharing behavior for different block sizes. Recall that our WU protocol does not issue updates (after the first one, of course) if no other processor is sharing the written block. Thus, in the beginning of the computation, processors have more time to write to blocks not yet shared by other processors when assuming the larger block sizes.

By relating the frequency of useless updates back to the source code, we can gain insight into the sharing patterns that produce them. Our analysis shows that widely-shared data are one common source of useless updates. A single processor that modifies data in a critical section sends an update message to each processor that has ever accessed the variables in the critical section, even though only one processor (the next one to enter the critical section) will need those updates. Shared counters and global work queues both produce numerous useless updates of this type. For example, 98% of the updates (with 64-byte cache blocks) in Mp3d can be found in a single routine that updates the cell array describing the state of the simulation space. More than 90% of those updates are proliferation updates.

Multiple consecutive writes to the same word may also cause a large number of proliferation updates, as each write except for the last one is guaranteed to be useless to the processors sharing the cache block. The sequential LU phase of Blocked LU generates useless updates due to this referencing behavior.

Under certain circumstances pair-wise sharing can be an important source of useless updates. If two processors share data and neither is the home node for the data, then each write to the shared data results in a useless (proliferation) update to the home node. Thus, even if the data is truly shared between the two processors, there will be a proliferation update per useful update. Consider SOR with 64-byte blocks as an example. Of the 290K proliferation updates in the program, 250K updates (which is 40% of all updates) are proliferation updates to the home node.

Load balancing schemes are another common source of useless updates. Work that migrates to an idle processor may leave behind a copy of the data in the cache, causing updates to retrace the path of migration. Roughly 46% of the updates in Barnes-Hut with 64-byte blocks occur in a single routine, and the vast majority of those updates are useless, proliferation updates caused by moving a body from one processor to another without flushing the cache. The more load imbalance occurs, the more bodies move among processors, creating new recipients for updates.

False sharing is another source of useless updates. For Em3d with 64-byte cache blocks, 27% of all updates are false updates; the percentage of false updates rises to

47% with an increase in block size to 256 bytes. `Mp3d`, `Blocked LU` and `CG` also suffer from false updates when using 256-byte blocks.

It is clear from these figures that the elimination of useless updates would improve WU performance tremendously. In some cases eliminating useless updates may be easy; our examination of the useless updates to the cell array in `Mp3d` uncovered the fact that certain values in the cell array were repeatedly modified but never used. By eliminating the useless code, we improved WI and WU running time performance by factors of 2 and 9, respectively, as well as reduced the number of updates by a factor of 2.5-3 depending on block size. Nonetheless, over 90% of all updates in the modified program are still useless, mostly proliferation updates. In our subsequent experiments we will use this modified `Mp3d`, referred to as `New Mp3d`, in place of the original.

The following sections evaluate the effect of more complicated techniques for reducing the number of useless updates. We first consider coalescing write buffers, a mechanism for combining multiple updates in a single message, and then study the impact of eliminating replacement updates. Finally, we consider techniques that eliminate proliferation updates.

6.4.2 Merging Updates with Coalescing Write Buffers

A coalescing write buffer [Jouppi, 1993; Thacker *et al.*, 1992] is simply a cache-block-wide buffer capable of merging writes to the same cache block. In the context of a WU protocol, this feature allows for a reduction in the number of updates propagated outside the processor. A coalescing write buffer is also useful for WI, since it usually reduces the average number of occupied buffer entries, and therefore induces fewer processor stalls. Note that a coalescing write buffer is slightly different from a write cache (e.g. [Dahlgren and Stenstrom, 1994]). A write cache sits between the cache and the memory bus and therefore has no effect on the write traffic going from the processor to the cache.

Our implementation of coalescing write buffers assumes 4 entries, each wide enough for a cache block.[1] We associate a dirty bit vector with each entry in a write buffer indicating the words that were written. If the processor writes to an address in a cache block that is already in the buffer, the new write is merged with earlier ones, and its dirty bit is set. If the processor writes to a block for the first time, a new buffer entry is allocated for the write. Unlike traditional write buffers, our coalescing buffer does not attempt to write its entries out immediately; it waits until there are 2 valid entries in the buffer or until it is forced to flush entries. When a write is issued from the buffer, only the dirty words are sent in the message. A coalesced update message locks out the cache of the receiver for the same number of cycles as it takes to transfer an entire cache block on the bus, subject to the constraint that the lockout time has to be at least the same as the number of dirty words in the coalesced message.

Our experimental results show that the addition of coalescing write buffers improves the execution time performance of both protocols; WI improves slightly, while WU

[1] In the previous experiments we assumed an 8-entry non-coalescing write buffer. We reduce the number of entries in the coalescing buffers because each entry takes up more chip area under coalescing.

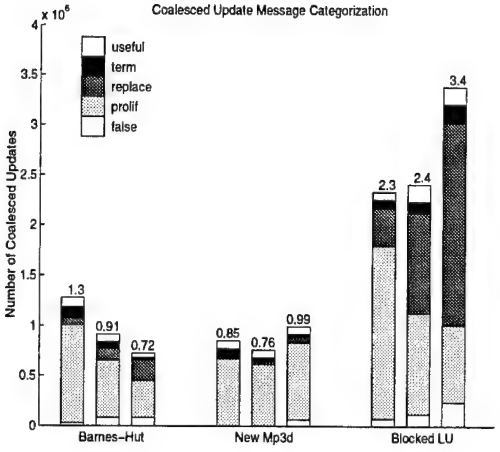


Figure 6.17: Categorization of coalesced updates for group 1 for blocks of 16 (left), 64 (center), and 256 (right) bytes.

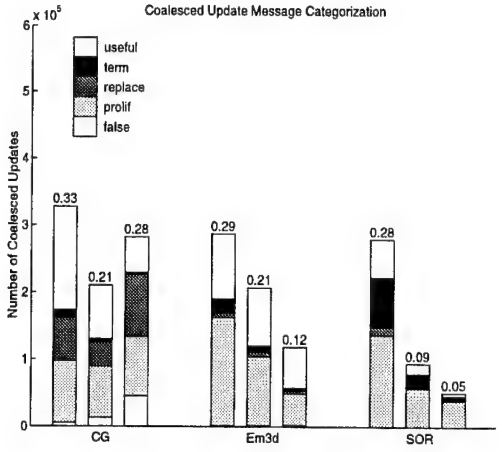


Figure 6.18: Categorization of coalesced updates for group 2 for blocks of 16 (left), 64 (center), and 256 (right) bytes.

improves significantly. The coalescing buffer's ability to combine multiple writes to a specific word does not provide significant performance gains for our applications, however. The only application that benefits from this characteristic of the coalescing buffers is `Barnes-Hut`. Under the WU protocol, this application achieves a 5-28% reduction in the total number of word updates sent across the network, depending on the block size.

Coalescing reduces the number of bytes transferred under WU by at least 20% in most cases. This reduction comes mainly from requiring fewer message headers for the coherence traffic.

The most significant gains provided by coalescing write buffers come from a major reduction in the number of coherence messages transferred through the network, with a corresponding reduction in the number of acknowledgements required. Figures 6.17 and 6.18 depict our categorization of the coalesced update transactions involved in our two groups of applications. The number on top of each column is the total number of coalesced update messages (in millions).

Our categorization of coalesced messages extends the definitions of useful and useless updates to apply to a collection of updated words (those included in a message) as opposed to individual words. We define the lifetime of a message to span from the receipt of the message until all the words updated by it are overwritten or the corresponding block is evicted from the processor's cache. Thus, a message is considered useful (true sharing) if at least one of the updates included in the message is useful. A false sharing message is one in which none of the updates is a true sharing update and at least one of the updates is a false sharing update. A message is classified as a proliferation message if all of the updates in the message are proliferation updates. Proliferation messages at the end of the program are classified separately as termination messages. A replacement update occurs when the block updated by the coalesced message is replaced from the

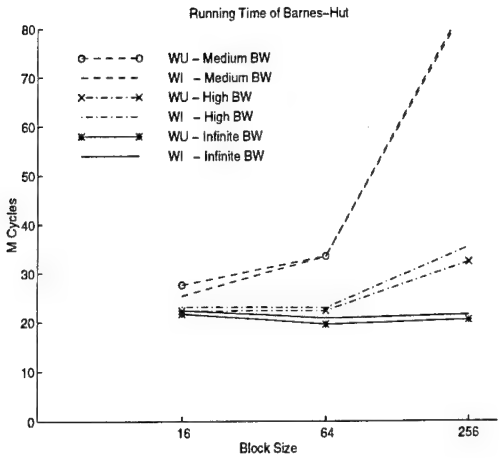


Figure 6.19: Running time of coalescing `Barnes-Hut`.

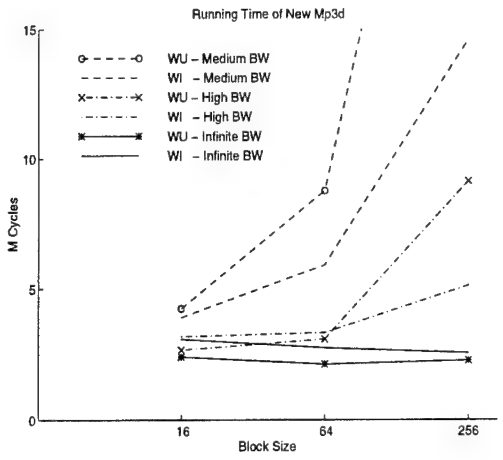


Figure 6.20: Running time of coalescing `New Mp3d`.

processor's cache and none of the updates in the message is a true sharing update.

Comparing these figures with the ones in the previous section (where each update corresponds to a message), we can see that, for all our programs, coalescing write buffers reduce the number of coherence messages under WU by at least 53%, except for `Em3d` with 16-byte blocks. All applications achieve more than 67% improvement in the number of coherence messages with the largest cache block we consider. `SOR` achieves the greatest reduction in the number of messages, 91% with 256-byte blocks, due to its perfect spatial locality.

We can also gather from the figures that coalescing changed the overall update behavior of two applications completely (`Barnes-Hut` and `Em3d`); for these applications, increasing the block size results in a reduction of the total number of update messages. This shows that these applications exhibit good spatial locality of writes to cache blocks. Three other applications (`New Mp3d`, `Blocked LU` and `CG`) exhibit good spatial locality of writes up to 64-byte blocks; larger blocks reverse the trend of decreased coherence traffic by significantly increasing the degree of sharing in the programs.

In terms of the percentage of useful update traffic with and without coalescing, our figures depict mixed results. `Em3d` exhibits an enormous increase in the percentage of useful coherence traffic with coalescing and the larger cache blocks, while `SOR` exhibits a significant decrease in that percentage for the same block sizes. The others applications exhibit roughly the same percentage of useful traffic, independently of whether coalescing is used. These results show that, although coalescing significantly reduces the number of messages and bytes sent across the network, there is plenty of room for further improvement as most of the coherence traffic is still useless.

As examples of the execution time improvement provided by coalescing, compare figures 6.19 and 6.3. Figure 6.3 shows the running time of `Barnes-Hut` with conventional write buffers as a function of bandwidth and block size, while figure 6.19 shows the same type of graph but assumes coalescing write buffers. Figure 6.20 shows the coalescing

running times for `New Mp3d`. Coalescing provides running time improvements under WU of as much as 43% for `New Mp3d` with infinite bandwidth and 256-byte blocks, and 19% for `Barnes-Hut` also with infinite bandwidth and 256-byte blocks. Other applications, such as `CG` and `Blocked LU`, also exhibit significant improvements in running time under WU and wide coalescing buffers with infinite bandwidth.

Performance improvements quickly degrade as we decrease the bandwidth available in the system, however. For instance, the execution time performance of `New Mp3d` under WU with coalescing, high bandwidth, and 256-byte blocks is a factor of 2 worse than with traditional write buffers. The reason for this effect is that, in the presence of a non-uniform distribution of memory accesses, coalescing write buffers may cause severe memory and network contention, as a result of the longer period of time resources remain busy per request. Larger cache blocks (and, therefore, potentially longer transactions) and lower bandwidth make this scenario worse. Shorter requests result in a greater degree of interleaving in network and memory utilization, allowing more processors to continually make forward progress. `New Mp3d` is again a good example. Under high bandwidth and 16-byte blocks, the performance of the program is 18% better with coalescing buffers than without them.

In short, when coalescing write buffers are effective at merging updates to large cache blocks, the result is long messages that occupy the network and memory for long periods of time and may cause contention. When coalescing write buffers are not effective at merging updates, wide buffers designed to hold large cache blocks waste chip space. These observations indicate that wide coalescing write buffers are not necessarily profitable.

6.4.3 Eliminating Replacement Updates

Under the largest cache block size we consider (256 bytes), most updates in `Blocked LU` are replacement updates. These updates are a result of a high replacement miss rate for shared blocks, which in turn is caused by the fact that processors access columns of the row-major-allocated shared matrix during parts of the computation. We can eliminate many of these updates by changing the data layout dynamically, so that the processor caches are more efficiently utilized. We experiment with *software caching* [Bianchini and LeBlanc, 1992], a technique originally proposed to reduce false sharing in WI protocols that, in effect, allows for a reorganization of the data.

Software caching consists of copying a range of virtual addresses to a different range of virtual addresses, allowing the application to determine when data copies are made, when local data is written back to the global data space, and how the data copies are organized. As a result, the coherence unit and the coherence protocol are both defined by the application. An application can tailor the unit of coherence to the data, thereby avoiding false sharing. It can also "schedule" writes to the global data space in order to alleviate contention. Finally, the application can change the data layout in order to reduce the number of replacement misses.

Software caching is only effective when processors can reuse the blocks they load into their caches; the technique should be avoided altogether when there is no possibility of

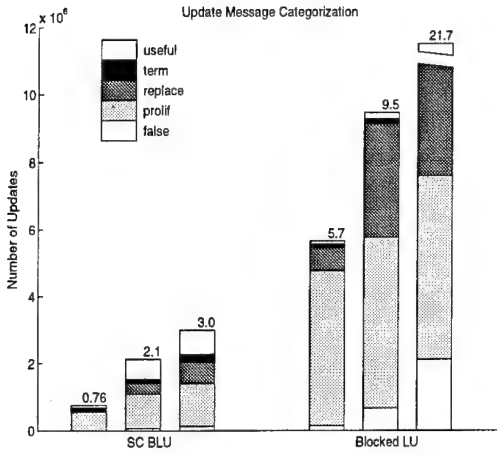


Figure 6.21: Categorization of updates for versions of `Blocked LU` for blocks of 16 (left), 64 (center), and 256 (right) bytes.

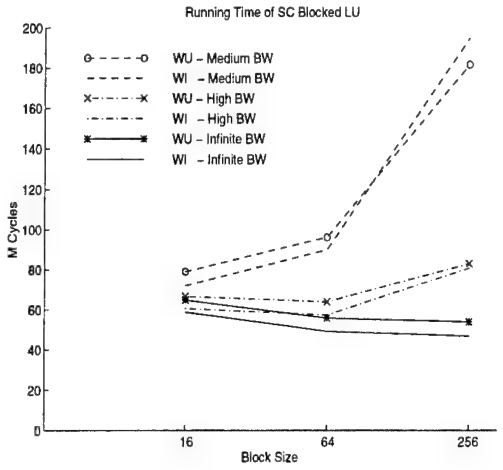


Figure 6.22: Running time of `SC Blocked LU` under pure WU protocol.

reuse. `Blocked LU` is the only application in our suite that suffers from a high percentage of replacement misses and exhibits the potential for reuse. We refer to the software caching version of the program as `SC Blocked LU`.

In `SC Blocked LU`, each processor makes a local, re-organized, copy of the data it needs, modifies the data as appropriate, and copies the result back to the original location when required for data sharing. Copy backs are scheduled in such a way that congestion in the network and memories is relieved. This technique incurs the overhead of making a local copy, but eliminates coherence transactions caused by false sharing and reduces the number of replacement misses in the program. (Note that some false sharing may occur when the data is copied back to its initial location, but this overhead is unavoidable if the algorithm requires the modified data back in the original location.)

Our simulation results show that software caching does reduce the network traffic of WU substantially. `SC Blocked LU` with 256-byte cache blocks sends about 3M update messages and 149MB of data across the network, while `Blocked LU` sends about 22M updates and 680MB of data with the same block size. Note that, for these cache block sizes, software caching reduces the number of bytes sent through the network substantially more than coalescing does, 78% against 33%.

As seen in figure 6.21, the reduction in the number of update messages sent by `SC Blocked LU` across the network comes from a 95% reduction in the number of replacement updates, a 94% reduction in false sharing updates, and a 77% reduction in proliferation updates. The percentage of useful updates increased from about 1% to 24% of the total number of updates, which is still somewhat small.

`SC Blocked LU` exhibits much better running time performance (shown in figure 6.22) than `Blocked LU` for the larger cache block sizes. Comparing the WI and WU executions of `SC Blocked LU`, we see that the impact of our program modifications is

greater for WI than for WU. The WU implementation suffers greatly from an increased critical path of execution, due to excessive update traffic backing up at the processor running the sequential LU phases of the algorithm. In section 6.4.5, we investigate whether coalescing can alleviate this problem.

6.4.4 Eliminating Proliferation Updates

In this section, we evaluate the performance of two strategies that can use invalidations to reduce the number of useless updates in update-based protocols. The first strategy consists of a hybrid WI/WU coherence protocol that allows for the static determination of the protocol to use on a per cache block basis. This strategy is an extension of the work presented in [Veenstra and Fowler, 1992] for bus-based multiprocessors and is referred to as our static hybrid protocol. The second strategy we study can also be considered a hybrid protocol in which processors dynamically self-invalidate cache blocks according to the update traffic directed to them. Protocols of this kind are usually referred to as *competitive update* protocols [Karlin *et al.*, 1988].

Static Hybrid

A static hybrid protocol must decide, for each cache block, whether that cache block should be managed with WU or WI. A *selection policy* is a rule for deciding whether a given cache block should use WU or WI. A block that uses WU is called a "WU-block" and a block that uses WI is called a "WI-block".

The motivation for using WU as opposed to WI for a certain cache block is that the read latency for the block may be reduced under WU, but only at the cost of some extra coherence traffic. Our selection policies estimate the read latency and the coherence overhead associated with each block under the different protocols. If a cache block has less coherence overhead with WU than read latency with WI, then that block should be included in the set of WU-blocks. In addition, it may be worthwhile to include a cache block in the set of WU-blocks even though that block has higher overhead with WU than with WI. The reason for this is that the updates can often be done in parallel with processor busy cycles, so the cost of the updates is hidden. Read misses, however, stall the processor. If too many cache blocks are included in the set of WU-blocks, however, then the additional update operations will increase network and memory contention. Thus, when deciding whether to include a cache block in the set of WU-blocks, the static hybrid selection policy must balance the potential reduction in cache misses against the potential increased cost of a cache miss.

The information needed by the selection policies is obtained from a simulation run that collects statistics about each cache block. The statistics needed to facilitate the selection of the WU-blocks are listed below.

- Rereads. The number of rereads for a cache block is the number of times a processor had to reread that block. If cache replacement effects are ignored, then using WU on a given cache block can eliminate all the rereads for that block.

- Extra updates. The number of extra updates is estimated by counting the total number of writes to a cache block and subtracting the number of invalidations required by WI for that cache block. Writes to a cache block that has not yet been shared are not included in the number of extra updates. This metric is intended to quantify the number of extra write operations that would be required if WU were to be used on that cache block.

The static protocol policies differ in how they use this information to select the WU-blocks. The first policy we study is conservative in selecting WU-blocks in that only those cache blocks that are estimated to have less coherence overhead using WU than read latency under WI are included in the set of WU-blocks. These blocks can be characterized by satisfying the following formula: (extra updates $\times$ cost of an update) $\leq$ (rereads $\times$ cost of a read). We will refer to the left and right-hand sides of this formula as WU-cost and WI-cost, respectively. We will refer to the sum of all WU and WI-costs of the blocks selected as Total WU-cost and Total WI-cost, respectively.

To the set of blocks chosen with our first selection policy, one can add blocks for which WU-cost > WI-cost. In this case, the rereads required by WI would be traded for the extra updates required by WU. Our second static hybrid policy selects blocks among the ones with the largest ratios (WI-cost / WU-cost) for inclusion into our first set of blocks. The policy includes just enough blocks to make Total WU-cost == Total WI-cost. More aggressive policies can be defined by allowing the number of WU-blocks to grow until the ratio between the two total costs exceeds a certain threshold. We study policies for which Total WU-cost is 10%, 20%, and 50% more than Total WI-cost. Our write stall statistics show that 25% of the non-overlapped cost of an update is a reasonable assumption for the update cost as observed by processors. We include more than one selection policy in our study in order to find a close approximation to the best policy.

Our experiments show that the second selection policy tends to deliver the best results overall for the static hybrid protocol. The effect of the protocol using this policy on the update traffic can be seen in figures 6.23 and 6.24. We can see in the figures that the static hybrid strategy is very successful at reducing the amount of (useless) coherence traffic in all applications, except for `CG` and `Em3d`. This reduction in the amount of useless traffic does not significantly increase the percentage of useful updates however, since the static hybrid protocol eliminates both useless and useful updates, but does not guarantee that more useless than useful updates are prevented. In fact, in two cases (`New Mp3d` and `SOR`), the static protocol exhibits a significant decrease in the number of useful updates.

Comparing the coherence traffic results for `SC Blocked LU` and `Blocked LU` under the static hybrid protocol, we see that software caching entails many fewer useless update transactions. Coalescing also compares favorably against the static hybrid technique; the former strategy generates as much as 38% less overall traffic than the latter for all applications and large block sizes.

The static hybrid protocol decreases the amount of coherence traffic associated with a pure WU strategy by simply not using updates for certain cache blocks, which may

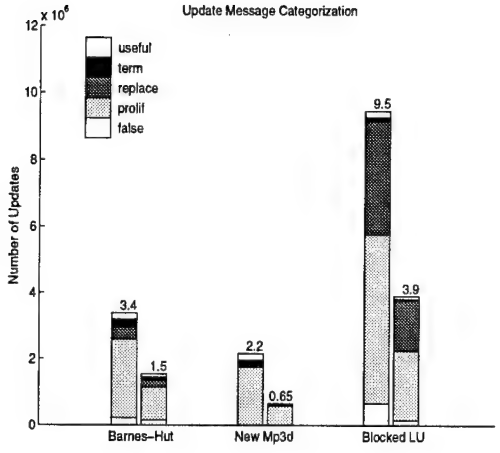


Figure 6.23: Categorization of updates for group 1 under pure WU (left) and the static hybrid protocol (right) for 64-byte blocks.

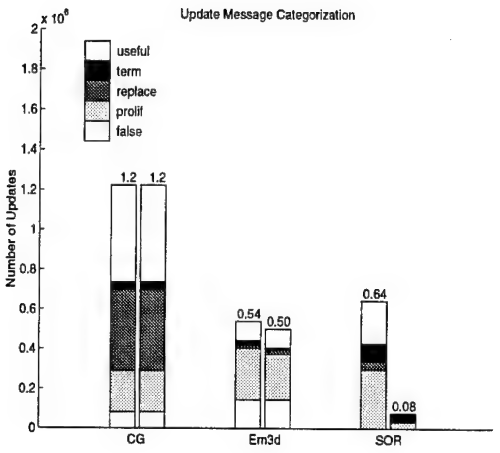


Figure 6.24: Categorization of updates for group 2 under pure WU (left) and the static hybrid protocol (right) for 64-byte blocks.

cause an increase in the miss rate. Figure 6.25 presents the read miss rates of our applications under WI (left) and the static hybrid strategy (right). Note that, in cases where this miss rate degradation is significant (such as `New Mp3d` and `Blocked LU`), running time performance suffers accordingly. However, even when the miss rates do not increase noticeably (as for `CG` and `Em3d`), programs may perform worse as a result of not using WU for performance-critical blocks. `Barnes-Hut` was the only application to achieve execution time improvements under the static hybrid strategy for relatively high bandwidths. Under high and infinite bandwidths and 64-byte blocks, the static hybrid approach improves running time performance by 10% in comparison to the pure WU protocol. The performance of `Barnes-Hut` remains worse than WI under the static hybrid protocol, however.

Our experience with the static hybrid strategy clearly demonstrates that its performance is heavily dependent on good initial estimates for the cost of updates. The problem is that it is very difficult to produce cost estimates that can be used effectively for all applications. These results suggest that the static hybrid protocol is of limited use for scalable multiprocessors, unless compilers can produce accurate estimates of the cost of updates for each application.

Dynamic Hybrid

The dynamic hybrid strategy is inspired by the coherence protocols of the bus-based multiprocessors using the DEC Alpha AXP21064 [Thacker *et al.*, 1992]. In these multiprocessors, each node makes a local decision to invalidate or update a cache block when it sees an update transaction on the bus. The decision depends on the presence of the block in the primary cache. (The contents of the secondary cache are a superset of the contents of the primary cache.) When the block is present in the primary cache, the

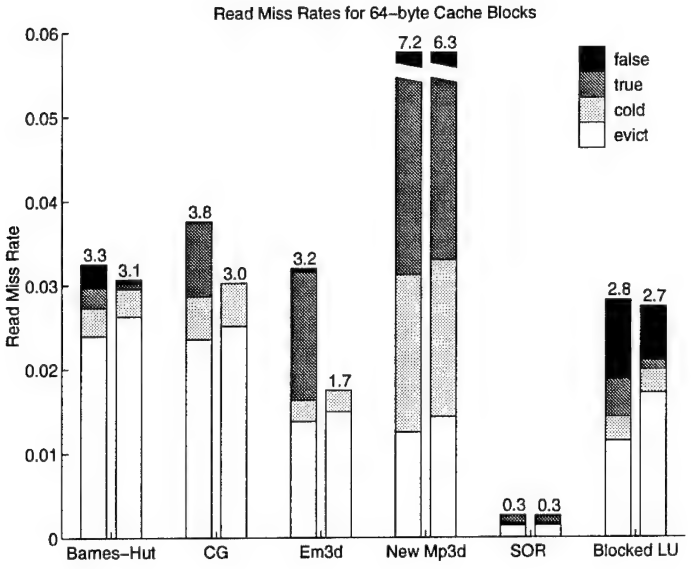


Figure 6.25: Miss rate under WI (left) and the static hybrid protocol (right) for 64-byte blocks.

cache controller updates the copy in the secondary cache and invalidates the copy in the primary cache. If the block is updated again before any reference by the processor, the cache controller invalidates the copy in the secondary cache. Thus, after at most two update transactions, an unused cache block is invalidated from the processing node.

As in [Dahlgren and Stenstrom, 1994], our implementation of this idea associates a counter with each cache block and invalidates the block when the counter reaches a threshold.[2] At that point, the node sends a message to the block's home node asking it not to send any more updates to the node. References to a cache block reset the counter to zero. We use counters with a threshold of 4 updates.

The dynamic hybrid protocol reduces the degree of sharing in applications significantly. Taking the average number of updates sent per write to shared data as our metric, we see that the dynamic hybrid strategy provides reductions in the degree of sharing for 256-byte blocks of as much as factors of 4.8, 2.3, 2.4, 8.4, 6.8, and 5.0 for `Barnes-Hut`, `CG`, `Em3d`, `New Mp3d`, `SOR`, and `Blocked LU` respectively, compared to pure WU and the other strategies for improving that protocol. Reducing the degree of sharing is an important characteristic of this strategy, as the running time performance of directory schemes based on a limited number of pointers degrades quickly when the hardware pointers are frequently exhausted.

Our categorization of coherence traffic in competitive protocols includes an extra category, drop updates, to account for the updates that cause blocks to be invalidated.

[2]The Alpha implementation uses the presence of the block in the primary or secondary cache as an implicit counter.

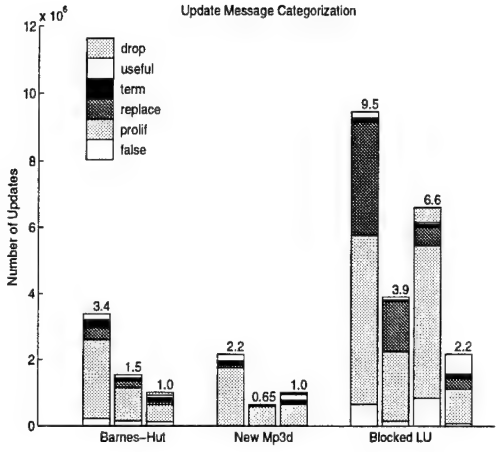


Figure 6.26: Categorization of updates for group 1 under pure WU (left), static (center), and dynamic (right) protocols for 64-byte blocks.

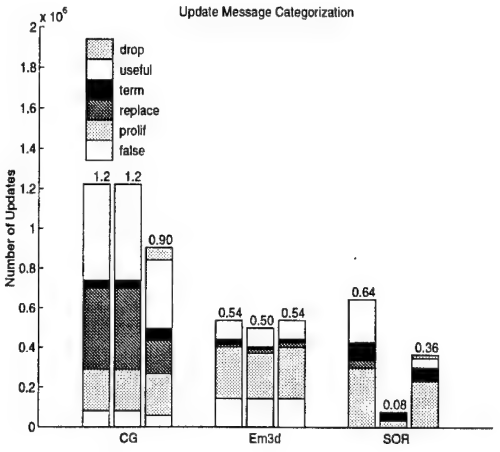


Figure 6.27: Categorization of updates for group 2 under pure WU (left), static (center), and dynamic (right) protocols for 64-byte blocks.

As seen in figures 6.26 and 6.27, the dynamic hybrid protocol is very effective at reducing the coherence traffic associated with the pure WU protocol. Depending on the application, either the dynamic or the static hybrid protocols generate the least amount of update traffic among all techniques we study. Comparing the traffic categorizations for `SC Blocked LU` against the ones for `Blocked LU` under the dynamic hybrid protocol, we find that the dynamic hybrid strategy entails significantly more (useless) updates. Coalescing generates less total traffic than the dynamic hybrid protocol for all applications; `CG` exhibits the largest difference: 12-31%, depending on the block size.

Software caching and the dynamic hybrid protocol are the most successful strategies for reducing the useless coherence traffic associated with the pure WU protocol, but, in some cases, the latter technique also reduces the number of useful updates. A reduction in the number of useful updates indicates that the protocol is forcing processors to drop copies of blocks they will need later. Dropping such blocks may or may not cause running time degradation, depending on a tradeoff between the number and impact of useless updates that would have resulted in not invalidating the blocks and the higher miss rate.

Figure 6.28 compares the read miss rates of our programs under WI (left) and under the dynamic hybrid protocol (right). Our categorization includes a new class of miss (labeled "Drop" in the figure), which occurs when a processor takes a miss on a block that was previously in the cache, but was invalidated when its counter reached the competitive threshold. The figure shows that `CG`, `New Mp3d`, `SOR`, and `Blocked LU` suffer significantly from bad invalidation decisions, while the two other programs are either mildly affected or not at all. The block size affects the drop miss rate of our applications in different ways. For the applications with excellent locality (`CG` and `SOR`), the number of drop misses decreases with an increase in block size, while for the other

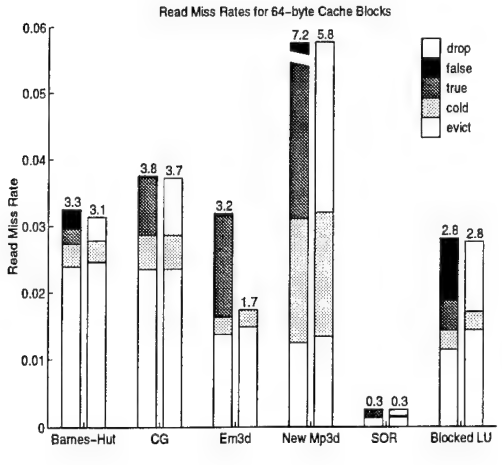


Figure 6.28: Miss rate under WI (left) and the dynamic hybrid protocol (right) for 64-byte blocks.

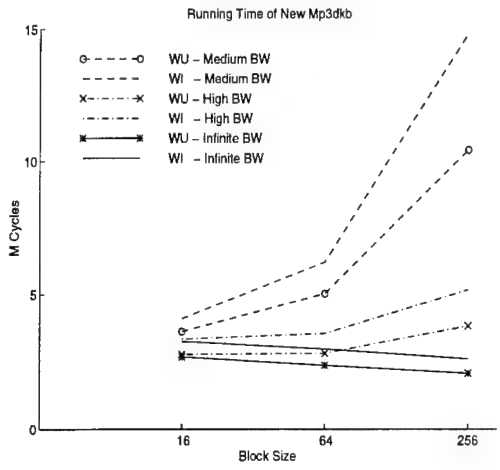


Figure 6.29: Running time of `New Mp3d` under dynamic hybrid protocol.

applications it either increases (`Em3d` and `Blocked LU`) or remains roughly the same (`New Mp3d` and `Barnes-Hut`).

Even though drop misses increase the miss rate of `New Mp3d`, this application obtains significant running time improvements under the dynamic hybrid protocol in comparison to pure WU: with 256-byte blocks, for instance, 2%, 19%, and 48% improvements were found with medium, high, and infinite bandwidth, respectively. A comparison against WI can be found in figure 6.29. The figure shows that `New Mp3d` performs somewhat better under the dynamic hybrid protocol than under WI for all bandwidth levels and block sizes. Albeit the good results for `New Mp3d`, the dynamic hybrid strategy does not achieve running time improvements for the other applications in our suite.

The poor miss rate performance of four of our applications under the dynamic hybrid protocol can be credited to the fact that, if programs exhibit a high degree of spatial locality of writes, 4 updates without intervention from the local processor is too small of a threshold, especially for the larger block sizes. One extreme option would have been to set the threshold to the number of words in a block plus 1. However, in the absence of write locality, this threshold would have entailed a large number of useless updates in the case of the large cache blocks. We opted for the threshold of 4 in order to reduce the number of useless updates. A combination of coalescing and the dynamic hybrid protocol should diminish the importance of the competitive threshold. We study this and other combinations of strategies in the next section.

6.4.5 Combining Techniques

Our results so far show that coalescing write buffers provide significant reductions in coherence traffic accompanied by consistent execution time improvements under the higher levels of bandwidth and moderately-sized cache blocks. The other techniques,

although also successful at reducing the total network traffic, don't always deliver running time improvements. In this section, we investigate combinations of coalescing with the other techniques.

Under the combination of coalescing and software caching, WU matches the running time performance of WI for `SC Blocked LU`, as the program improves by 8-16% under WU, depending on the block size, and the WI performance remains roughly unaltered.

The combination of coalescing and the static hybrid protocol improved on the execution time performance of the static hybrid technique in isolation in all cases. The combination of coalescing and the dynamic hybrid protocol (still with a competitive threshold of 4 updates) frequently improves the running time of the dynamic hybrid strategy in isolation, while it occasionally improves on the running time of coalescing. Coalescing eliminates the problems associated with the specific value of the threshold, while the dynamic hybrid optimization reduces the degree of sharing in the program. For programs such as `New Mp3d` and `CG`, the reduced sharing markedly improves the execution time performance of coalescing for the larger cache blocks under the practical levels of bandwidth.

In summary, all techniques we studied and their combinations were very successful at reducing the amount of useless coherence traffic generated by pure WU protocols. Software caching (when applicable) and the dynamic hybrid protocol stand out as being able to significantly increase the percentage of useful updates in applications. Coalescing write buffers provided the greatest reductions in the total number of bytes transferred as well as the greatest improvements in running time. In only a few instances did another technique or combination of techniques outperform coalescing. The major performance problem for coalescing buffers occurs when cache blocks are very large and the bandwidth in the system is not extraordinarily high.

6.4.6 Potential for Additional Improvements to Update-Based Protocols

The traffic categorizations presented in the previous section clearly show that useless updates dominate the coherence traffic of our applications, even when techniques intended to reduce this type of traffic are applied. In this section, we comment on the potential for further reductions in the percentage of useless updates.

Our analysis of the sources of updates in applications shows that a large number of proliferation updates stems from a round-robin assignment of pages to processors. The problem with this type of mapping is that it is often the case that the home node is not one of the processors sharing the blocks for which it receives updates. Thus, a simple strategy that can further reduce the number of proliferation updates is to assure that one of the processors sharing a block is the home node for that block. In fact, the processor that writes to the block the most should be made the home of the block. Page placement and migration techniques, such as presented in [Chandra *et al.*, 1994; Marchetti *et al.*, 1995], can be used to implement this strategy successfully in many cases. If applied to `SOR`, for instance, this strategy would have eliminated about half of the useless updates in the program.

Another way in which proliferation updates can be eliminated is by combining writes to the same words in software. This optimization can be implemented at the compiler level, by having the compiler use registers for shared data writes inside of critical sections, and only issuing writes to memory at the end of the section. Our experiments with `SC Blocked LU` implemented this strategy at the application level; throughout the program writes to global data were only issued when their final values had been computed. This scheme proved extremely useful for improving `Blocked LU`, but did not appear applicable to the other applications we studied.

Flushing widely shared cache blocks at the end of critical sections can also greatly reduce the number of proliferation updates. Centralized counters, locks, and barriers are examples of data structures that cause a large number of useless updates. Distributing those data structures should improve execution time performance even more significantly however, as it would avoid the increased read latency generated by block flushes.

6.5 Protocol Performance on Future Multiprocessors

Our results so far have shown that WU performs at least as well as WI for *all* applications and combinations of bandwidth and block size we consider, provided that techniques for reducing the amount of useless update traffic are applied. Under infinite bandwidth, the running time performance advantage of the update-based protocols over WI ranges from a few percent for the applications dominated by replacement misses (`Barnes-Hut` and `CG`) and applications with extremely small miss rates (`SC Blocked LU` and `SOR`) up to 23% for the applications dominated by coherence misses (`New Mp3d` and `Em3d`).

Note however that our previous results are based on current architectural assumptions, which do not favor update-based protocols in many respects: replacement misses (as opposed to sharing-related misses) dominate the miss rate of many applications, network and memory latencies are relatively low considering the latest advances in superscalar microprocessors, the highest practical level of bandwidth is relatively low considering the amount of data an update-based protocol has to tackle, and our memory bus assumptions are such that a coalesced update with one dirty word takes as long to complete as if all the words in the (possibly large) cache block were dirty.

We now extrapolate the current architectural trends in order to explore the WI versus WU issue in the context of a more aggressive scalable multiprocessor design. In order to quantify the impact of future architectures on protocol performance, we double the memory and network latency and bandwidth, simulate a memory bus that can handle variable length update operations, and increase the size of caches to 128K bytes. This increase in cache size is intended to approximate the availability of relatively large second-level caches, as opposed to large primary caches.

Figure 6.30 presents the WI and WU running times of our applications on a next-generation architecture with coalescing write buffers and 64-byte blocks. For all of our applications, the difference in running time between the protocols increased under the more aggressive assumptions of this architecture. Comparing our previous results for high bandwidth and coalescing against these new results, we see that the running time

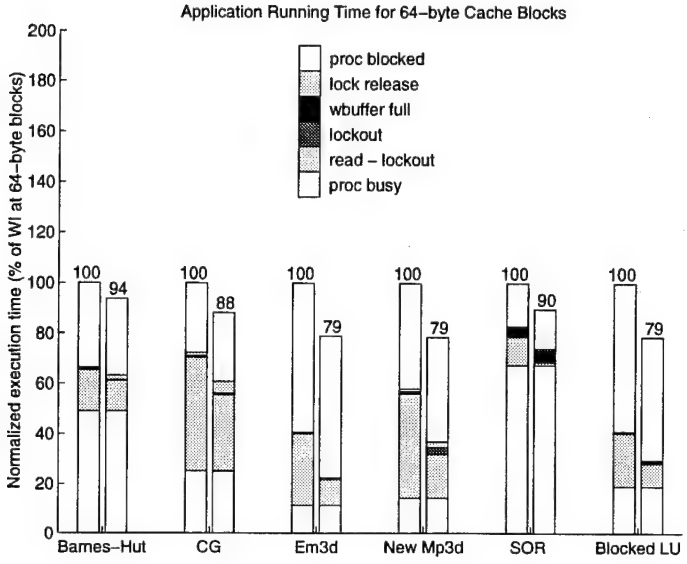


Figure 6.30: Running time of WI (left) vs. WU with coalescing (right) on next-generation architecture.

improvement of `CG` for 64-byte blocks goes from 8% to 12%, while the improvements of `New Mp3d`, `Blocked LU`, and `Em3d` go from 8% to 21%, 5% to 21%, and 17% to 21%, respectively. These results suggest that the architectural trends we have been observing should increase the performance advantage of WU over WI significantly in the future.

6.6 Summary

Our simulations of WU and WI coherence protocols for scalable multiprocessors showed that, although WU produces a lower miss rate, the enormous network traffic generated by WU degrades the running time of some of our applications. Even infinite bandwidth is not enough to enable WU to significantly outperform WI on all of our applications. We found the cause of the poor WU performance to be that the excessive number of update transactions in a pure WU protocol generates network and memory congestion, which is reflected in various forms of performance degradation.

To alleviate the network traffic generated by WU, we classified updates into useful and useless categories, and showed that a vast majority of the updates are useless; in most cases, useless updates are more than 90% of the total number of updates. By relating the useless updates back to the source code, we determined the application characteristics that cause useless updates and evaluated several software and hardware techniques for eliminating them. These techniques include coalescing write buffers, dynamic and static hybrid protocols, and software caching.

We studied the isolated and combined effect of these techniques on our categorization of the coherence traffic and on application execution time. Our results showed that software caching (when applicable) and the dynamic hybrid protocol are the most successful strategies for eliminating useless updates, and that software caching and coalescing write buffers produce the lowest amount of traffic by merging multiple updates in a single message. In terms of execution time, coalescing write buffers exhibit the most consistent improvements. Coalescing improves WU enormously, but only slightly improves WI. Wide coalescing write buffers were shown unnecessary for WU and may even cause serious performance degradation in the presence of relatively low bandwidth. The combination of coalescing and the dynamic hybrid protocol also performed well. Coalescing improves the dynamic hybrid strategy by reducing the importance of the specific value of the competitive threshold, while self-invalidating of cache blocks reduces the degree of sharing in application programs.

Although the techniques we considered significantly reduce the useless traffic associated with update-based protocols, a large number of useless updates remains. Based on that observation, we suggested several directions for further eliminating useless traffic in update-based protocols, such as flushing widely shared cache blocks at the end of critical sections to reduce the number of proliferation updates found in applications.

Finally, our experiments showed that WU (with optimizations) performs better than our WI implementation for all of our applications, confirming previously published results. Current architectural trends (faster processors, longer latency, higher bandwidth) significantly magnify the performance difference between the two types of protocols in favor of WU.

7 Conclusions

In this dissertation we explored ways to exploit bandwidth to reduce the average memory access time in scalable multiprocessors. Our studies considered situations where the remote memory access time is dominated by contention, and where the hardware provides high remote access bandwidth and the remote access time is dominated by the duration of the remote (unsaturated) request/access/reply sequence of operations.

The main contributions of this thesis are:

- The design and analysis of eager combining, a hardware coherence protocol that uses data distribution and request combining to increase the effective network and memory bandwidth. Eager combining was shown to achieve significant improvements in running time performance (as much as a 4-fold improvement), as a result of the increase in effective bandwidth. We also showed that eager combining consistently outperforms software broadcasting on our application programs.

- The design and analysis of software interleaving, a software implementation of memory interleaving also intended as a strategy for increasing the effective bandwidth available to processors applied to matrix computations. Software interleaving was shown to dramatically reduce memory contention, performing much better than row-major allocation on large-scale machines. When compared to logarithmic broadcasting, we showed that the best choice of technique depends both on the type of synchronization used and the number of processors.

- An analysis of large cache blocks as a way of reducing average memory access times in the presence of high remote access bandwidth. Using execution-driven simulation we found that the block size that produces the lowest mean cost per reference usually falls between 32 and 128 bytes. Our analytical model of large cache blocks suggests that the improvements in miss rate beyond 128-byte blocks are usually too small to offset the corresponding increase in miss penalty. Our model also shows that, although the few applications with excellent spatial locality and limited sharing may be able to exploit larger cache blocks, both the remote access latency and bandwidth must be much greater than we can expect in the foreseeable future for these blocks to improve performance noticeably.

- The design and analysis of hybrid prefetching, a technique combining hardware and software support for stride-directed prefetching. Our comparison of large

cache blocks, and sequential and hybrid prefetching showed that the latter technique performs at least as well as the others. In fact, given sufficiently high bandwidth and regular memory addressing, hybrid prefetching can perform as well as software prefetching.

- An analysis of write update as a protocol to reduce the average memory access time in the presence of high remote access bandwidth. Our simulation results showed that even infinite bandwidth is not enough to enable pure write-update to outperform write-invalidate consistently. The excessive number of useless update transactions in a pure WU protocol was shown to generate various forms of performance degradation. Our results showed that software caching and the dynamic hybrid protocol are the most successful strategies for eliminating useless updates, and that software caching and coalescing write buffers produce the lowest amount of traffic by merging multiple updates in a single message. We suggested several directions for further eliminating useless traffic in update-based protocols.

Our work led to two overall conclusions. The first is that techniques for increasing the effective remote access bandwidth available to processors, in particular software interleaving and eager combining, can lower average access times significantly. Software interleaving has much lower cost but a more restricted applicability than eager combining.

The second conclusion is that high remote access bandwidth is not a panacea. Techniques for overlapping communication and computation we consider suboptimal today will probably continue to be so with higher bandwidth, unless they are modified somewhat. Our work has determined the limitations of these techniques when bandwidth is not a major constraint and includes a few attempts at modifications of the techniques.

Our contributions and conclusions apply directly to scalable cache-coherent multiprocessors; the tradeoffs involved in other styles of shared memory multiprocessing are roughly similar to the ones we have considered, but a few differences exist. In particular, in software distributed-shared-memory systems, the size of the coherence unit (usually the same as a virtual memory page) is likely to remain large; not only as a result of the extremely high communication latencies of these systems, but also because the machine nodes' local memories are used as very large fully-associative caches, eliminating pollution problems. Furthermore, these systems alleviate the false sharing problem by using relaxed consistency models that allow for multiple concurrent writers per page.

The fact that software distributed-shared-memory systems defer the transfer of all forms of coherence messages to synchronization points leads to different tradeoffs when considering whether to use invalidate or update-based protocols for coherence. Delaying messages allows for eliminating useless updates and for coalescing several updates over a large window of time.

Clearly more work remains to be done if the bandwidth to be provided by future multiprocessor architectures is to be put to good use. Some of this additional work can follow from specific contributions made in this dissertation:

- The advent of highly superscalar processors with clock speeds of several hundred MHz is likely to generate new interest in contention alleviation techniques. A potential avenue for investigation is the performance impact of software interleaving and eager combining in the context of these processors.

- In chapter 5 we assumed an ideal implementation of software prefetching. However, software prefetching compilers have limitations, such as an inability to detect potential replacement misses caused by dynamically-allocated data. A study of the combination of sequential or hybrid prefetching and software prefetching is another opportunity for extending our work.

- In chapter 6 we suggested several techniques that can further reduce the amount of useless traffic in update-based coherence protocols. Additional research on these techniques is very likely to lead to significant performance improvements.

- Parallel programs do not require high remote access bandwidth constantly; executions alternate between phases of high and low bandwidth requirements [Boothe and Ranade, 1993]. Another interesting piece of research could investigate techniques for adjusting how "aggressively" latency-tolerant techniques are applied during the execution of a program, depending on estimates of bandwidth utilization.

A Algorithms for Categorizing Communication Traffic

The existence of multiple copies of the same data in different caches poses the *cache coherence problem.* There are two common classes of coherence protocol for cache-coherent shared-memory multiprocessors: write-update protocols (WU) and write-invalidate protocols (WI) Both WI and WU schemes have the potential to introduce communication due to a mismatch between the hardware unit of coherence and the data structures manipulated by the program. As an example of such a mismatch, consider a situation where two processors read and write *different* portions of a multi-word cache block. In this case, the coherence protocol has to maintain the copies of the block consistent, even though each processor will never use the other processor's changes [1]. Relatively small caches can also cause unnecessary communication, due to replacement misses in the absence of good temporal locality.

An excessive amount of unnecessary traffic may significantly slow programs down by increasing the number and duration of processor stalls. After detection, useless communication can be eliminated either by using efficient coherence protocols, clever compilers, or program restructuring techniques.

This appendix is describes algorithms for detecting and categorizing useless communication under WI and WU coherence protocols. We focus on classifying the major sources of communication exhibited by programs simulated under these protocols. In section A.1 we extend a well-known algorithm for classifying cache misses, in order to account for finite-sized caches under invalidate-based protocols. In addition, we introduce a new algorithm that accurately characterizes update transactions under a write-update protocol. Extensions of this algorithm consider coalescing write buffers and competitive update strategies.

Most of the work dealing with multiprocessor communication does not seek to accurately characterize the source of the observed traffic. Notable exceptions, such as [Gupta and Weber, 1992], have focused on measuring the amount of each type of communication (requests, data, and coherence), and coupling these measures with information about the reference and sharing patterns of application programs. Studies of multiprocessor communication have invariably been concentrated on WI protocols.

[1] The term *false sharing* has been used to describe this type of interference in write-invalidate protocols, e.g. [Eggers and Jeremiassen, 1991; Torrellas *et al.*, 1994].

Although very few papers have studied the usefulness of the communication traffic observed when running parallel programs, there has been some research on developing algorithms that attempt to characterize false sharing under invalidate-based protocols [Dubois *et al.*, 1993; Eggers and Jeremiassen, 1991; Torrellas *et al.*, 1994]. Dubois *et al* [Dubois *et al.*, 1993] propose a scheme that delays classifying a miss until the subsequent invalidation of the block missed on, or the end of the program (whichever happens first). They show that their scheme accurately captures the intuition behind false sharing and remedies the problems encountered in previous approaches. Our algorithm for classifying cache misses extends the one presented by Dubois *et al* to tackle replacement misses. None of the above approaches attempt to incorporate eviction miss detection into the classification algorithm.

Write-update classification schemes have received almost no attention. Khera *et al* [Khera *et al.*, 1993] present an analysis of false sharing that attempts to characterize false sharing independently of architecture. However the approach is statistical in nature and encompasses several assumptions; such estimates can be extremely inaccurate in practice.

A.1 Algorithms

In this section we present the algorithms for classifying the major sources of communication under both WI and WU coherence protocols. For WI protocols the major source of communication is data transfers caused by cache misses, while WU protocols also incur a significant amount of traffic caused by update transactions (and their associated acknowledgements).

We assume a simulator structure that has separate routines for handling read misses, read hits, write misses, write hits and invalidations. The code we present is an addition to the code implementing those routines.

```
void read_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (Comm[proc_id, word]) {
    Essential[proc_id, block_id] = True;
    foreach wrd in block_id
      Comm[proc_id, wrd] = False;
  }
}

void read_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (Infinite[proc_id, block_id]) {
    /* Should have been in the cache */
    M_evict++;
    Classified[proc_id, block_id] = True;
  } else {
    /* Don't know if miss is useful yet */
    Essential[proc_id, block_id] = False;
    Classified[proc_id, block_id] = False;
    read_hit_class(proc_id, block_id, word);
  }
}

void write_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!Dirty[proc_id, block_id])
    M_excl++;
  else
    read_hit_class(proc_id, block_id, word);
  foreach proc not sharing block_id
    Comm[proc, word] = True;
}

void write_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  read_miss_class(proc_id, block_id, word);
  foreach proc not sharing block_id
    Comm[proc, word] = True;
}
```

Figure A.1: Classification of cache misses under a WI protocol.

```
void complete_miss_request_class(proc_id, block_id)
{
  Infinite[proc_id, block_id] = True;
}

void invalidate_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  Infinite[proc_id, block_id] = False;
  if (!Classified[proc_id, block_id])
    classify(proc_id, block_id);
  Comm[proc_id, word] = True;
}

void replacement_class(proc_id, block_id)
int proc_id, block_id;
{
  if (!Classified[proc_id, block_id])
    classify(proc_id, block_id);
}

void classify(proc_id, block_id)
int proc_id, block_id;
{
  if (Present[proc_id, block_id]) {
    if (!Cold[proc_id, block_id]) {
      M_cold ++;
      Cold[proc_id, block_id] = True;
    } else
      if (Essential[proc_id, block_id])
        M_true ++;
      else M_false ++;
        Classified[proc_id, block_id] = True;
  }
}

void end_program_class()
{
  foreach proc
    foreach block
      if (!Classified[proc, block])
        classify(proc, block);
}
```

Figure A.2: Classification of cache misses under a WI protocol - Cont.

A.1.1 Data Traffic Under a WI Protocol

Our algorithm for invalidate-based coherence is a simple extension of the one presented in [Dubois *et al.*, 1993]. We categorize cache misses in terms of the reference and sharing behavior causing them. We identify four basic categories of misses:

- **Cold start misses.** A cold start miss happens on the first reference to a block by a processor.
- **True sharing misses.** A true sharing miss happens when a processor references a word belonging in a block it had previously cached but has since been invalidated, due to a write by some other processor to the same word.
- **False sharing misses.** A false sharing miss occurs in roughly the same circumstances as a true sharing miss, except that the word written by the other processor is not the same as the word missed on.
- **Eviction misses.** An eviction (replacement) miss happens when a processor replaces one of its cache blocks with another one mapping to the same cache line and later needs to reload the replaced block.

Cold start and true sharing misses are necessary for the correct execution of the program, so they can be thought of as useful (essential) misses, while false sharing and eviction misses represent shortcomings of the architecture and/or the program and thus are considered useless misses. Figures A.1 and A.2 present the algorithm for classifying cache misses under a WI protocol.[2] Misses are classified at the end of a block's lifetime in the cache, which happens as a result of an invalidation, a replacement, or the termination of the program. An exception to this rule is eviction misses (and exclusive requests) which are classified the moment a block is brought into the cache, since the status of the block cannot change until the end of its lifetime.

The main data structures used in this algorithm are six two-dimensional arrays indexed by processor identification numbers and cache block numbers, and a matrix (**Comm**) indexed by processor numbers and word addresses. The functionality of each data structure is as follows:

Comm. This bit array saves information about writes to each of the words in the system. When a processor writes a word, all remaining processors have their **Comm** bit set for that word. When a processor accesses a word, it checks to see if its **Comm** bit is set. If this is the case and the miss that brought the block into the cache is not a cold miss, there was useful communication between processors.

[2]Note that our algorithm includes a fifth category, exclusive request transactions. An exclusive request operation (caused by a write to a read-shared block already cached by the writing processor) is not strictly a cache miss, although the processor may have to stall until it receives ownership of the block.

Cold. This bit array is used to classify cold start misses. A **Cold** bit is set when the first miss by a processor on a certain block is detected by the algorithm.

Essential. When a miss occurs it is marked as non-essential. Future references to the cache block may change this characterization to essential, if the processor accesses a word whose **Comm** bit is set.

Classified. This array is used to make sure that a miss classified as an eviction miss at the beginning of a block's lifetime does not get reclassified later.

Infinite. This array represents the caches' contents assuming that caches are infinite. A miss on a block present in the processor's cache according to the **Infinite** array determines a replacement miss.

Present and **Dirty**. These arrays represent the present and dirty bits associated with each block in the caches. These arrays must already exist in any simulation of caches.

A.1.2 Data and Update Traffic Under a WU Protocol

Our algorithm for classifying the dominant sources of communication traffic under write-update categorizes both cache misses and update transactions. Since, in a pure write-update protocol, cache blocks are never invalidated, cache misses can only be of two kinds: cold start and eviction misses. We have identified five categories of update transactions: true sharing, false sharing, proliferation, replacement, and termination. True sharing updates are useful, since they are necessary for the correctness of the program. The other categories comprise the set of useless updates. More specifically,

- **True sharing updates.** The receiving processor references the word modified by the update message before another update message to the same word is received.
- **False sharing updates.** The receiving processor does not reference the word modified by the update message before it is overwritten by a subsequent update, but references some other word in the same cache block.
- **Proliferation updates.** The receiving processor does not reference the word modified by the update message before it is overwritten, and it does not reference any other word in that cache block either.
- **Replacement updates.** The receiving processor does not reference the updated word until the block is replaced in its cache.
- **Termination updates.** A termination update is a proliferation update that occurs at the end of the program.

Figures A.3 and A.4 present the algorithm for classifying update transactions under a WU protocol. In this algorithm, misses are classified at the moment they happen,

```
void read_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  Updused[proc_id, word] = True;
  foreach wrd in block_id
    Refd[proc_id, wrd] = True;
}

void read_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!Cold[proc_id, block_id]) {
    Cold[proc_id, block_id] = True;
    M_cold ++;
  } else M_evict ++;
    read_hit_class(proc_id, block_id, word);
}

void write_hit_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  read_hit_class(proc_id, block_id, word);
}

void write_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  read_miss_class(proc_id, block_id, word);
}

void recv_upd_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!First[proc_id, word])
    First[proc_id, word] = True;
  else
    if (Updused[proc_id, word])
      U_true ++;
    else if (Refd[proc_id, word])
      U_false ++;
    else if (end_of_program)
      U_term ++;
    else U_prolif ++;
  Updused[proc_id, word] = False;
  Refd[proc_id, word] = False;
}
```

Figure A.3: Classification of data and update transactions under a WU protocol.

```
void replace_updates(proc_id, block_id)
{
  foreach word in block_id
    if (First[proc_id, word]) {
      if (Updused[proc_id, word])
        U_true ++;
      else U_replace ++;
        First[proc_id, word] = False;
    }
}

void end_program_class()
{
  end_of_program = True;
  foreach proc, block, and word
    recv_upd_class(proc, block, word);
}
```

Figure A.4: Classification of data and update transactions under a WU protocol - Cont.

since their status cannot change afterwards. Update messages are classified at the end of an update's lifetime, which happens when the update is overwritten by another update to the same word, when the cache block containing the updated word is replaced, or when the program ends.

The most important data structures used for the classification of update transactions are three two-dimensional arrays of bit flags indexed by the processor identification number and the word referenced. The functionality of each data structure is as follows:

Updused. When a processor reads or writes a word in a cache block, the corresponding entry of **Updused** is set, signifying that the word has been used. Upon receipt of an update for a word, the algorithm checks the corresponding entry in this array. If it is set then the previous update is classified as a true sharing update.

Refd. When a processor accesses a cache block all words in the block are marked as referenced by setting the corresponding entries of **Refd**. When an update transaction is found to be useless, this array is examined to decide between the remaining categories. If the corresponding bit in the array is set, it implies that some other word in the block has been referenced and, therefore, that the block is undergoing active false sharing.

First. This array allows us to postpone classification of update transactions until there have been at least two updates to the same word.

A.1.3 Update Transaction Categorization with Coalescing Buffers

In this section, we show how to adapt our algorithms to classify the major sources of communication for hardware that coalesces multiple updates into a single message.

Coalescing [Jouppi, 1993] merges writes to the same cache line and only sends them out when the number of entries present in the coalescing buffer exceeds a certain value. The problem introduced with coalescing is that updates are delivered in groups so the messages seen by the communication media are decoupled from the individual updates sent by the processors. If our goal is to classify communication (i.e. the messages sent by processors) the algorithm of figures A.3 and A.4 is not sufficient. It is however relatively easy to extend the algorithm to account for coalescing. First, we need to extend the definitions of useful and useless updates to apply to a collection of updated words (those included in a message) as opposed to individual words. Second, we must consider the lifetime of coalesced update messages. We define the lifetime of a message to span from the receipt of the message until all the words updated by it are overwritten or the corresponding block is evicted from the processor's cache. Our extended set of definitions is as follows:

- **True sharing messages**. At least one of the individual updates included in the coalesced message is a true sharing update.
- **False sharing messages**. None of the individual updates included in the message is a true sharing update and at least one of them is a false sharing update.
- **Proliferation messages**. All of the updates in the coalesced message are proliferation updates.
- **Replacement messages**. The updated block is replaced from the processor's cache and none of the updates in the message is a true sharing update.
- **Termination messages**. Proliferation messages at the end of the program are classified separately as termination messages.

In figures A.5 and A.6 we present the routines that classify coalesced messages (and individual word updates) in the presence of a coalescing write buffer.[3] The routines dealing with cache hits and misses remain the same as defined in section A.1.2. We also keep the same data structures used in that section. However, our definition of message lifetime calls for a new data structure, **msg**, a two-dimensional array of records indexed by a processor number and a word address. This array is responsible for keeping track of the different messages sent to each cache block and their classification throughout the computation. The updates received in each message are identified by having the same "time stamps". The classification of each message proceeds over the lifetime of the message and obeys the priorities defined by the **TRUE_U**, **FALSE_U**, **TERM_U**, and **PROLIF_U** macros. More specifically, the values representing the classification of each message can only grow larger during the message's lifetime. Replacement messages override these priorities.

[3]Due to space constraints, we omit the routines for classifying messages when blocks are replaced and when the program ends.

```
#define TRUE_U   3
#define FALSE_U  2
#define TERM_U   1
#define PROLIF_U 0

void Upgrade_msg(proc_id, word, flag)
int proc_id, word, flag;
{
  time = msg[proc_id, word].time;
  /* Update class of words sent in msg */
  foreach wrd in block
    if (time == msg[proc_id, wrd].time)
      if (msg[proc_id, wrd].class < flag)
        msg[proc_id, wrd].class = flag;
}

void class_individual_upd(proc_id, word)
int proc_id, word;
{
  if (Updused[proc_id, word]) {
    U_true ++;
    Upgrade_msg(proc_id, word, TRUE_U);
  } else if (Refd[proc_id, word]) {
    U_false ++;
    Upgrade_msg(proc_id, word, FALSE_U);
  } else if (end_of_program) {
    U_term ++;
    Upgrade_msg(proc_id, word, TERM_U);
  } else {
    U_prolif ++;
    Upgrade_msg(proc_id, word, PROLIF_U);
  }
}
```

Figure A.5: Classification of update transactions under WU with coalescing.

```
void class_coalesced_msg(proc_id, word)
int proc_id, word;
{
  switch (msg[proc_id, word].class) {
    case TRUE_U:   Msg_true ++; break;
    case FALSE_U:  Msg_false ++; break;
    case TERM_U:   Msg_term ++; break;
    case PROLIF_U: Msg_prolif ++;
  }
}

void recv_msg_class(proc_id, block_id, words, nwr)
int proc_id, block_id, *words, nwr;
{
  m_recv[proc_id] ++;
  for (i = 0; i < nwr; i ++) {
    if (First[proc_id, words[i]]) {
      class_individual_upd(proc_id, words[i]);
      time = msg[proc_id, words[i]].time;
      remain = 0;
      /* Classify prev msg if about to die */
      foreach wrd in block_id
        if (time == msg[proc_id, wrd].time)
          remain ++;
        if (remain == 1)
          class_coalesced_msg(proc_id,words[i]);
    } else First[proc_id, words[i]] = True;
    /* Initialize data for msg just received */
    msg[proc_id,words[i]].time = m_recv[proc_id];
    msg[proc_id,words[i]].class = PROLIF_UPD;
    Updused[proc_id, words[i]] = False;
    Refd[proc_id, words[i]] = False;
  }
}
```

Figure A.6: Classification of update transactions under WU with coalescing - Cont.

A.1.4 Data and Update Traffic Categorization for Competitive Protocols

In this section we present an algorithm for classifying the traffic entailed by a hybrid WU+WI protocol inspired by the coherence protocols of the bus-based multiprocessors using the DEC Alpha microprocessor. In these multiprocessors, each node makes a local decision to invalidate or update a cache block when it sees an update transaction on the bus. We extend our categorization of update transactions to include a separate category of useless traffic for the update (coalesced message) that causes a block to be invalidated. We refer to updates (messages) in this new category as **Drop** updates (messages).

Figure A.7 presents the necessary modification to the **read_miss_class** routine (refer to figure A.3) and two additional routines to be called when a block is voluntarily dropped from the cache. The bit array **Dropped** records the fact that the corresponding block has been invalidated in the processor's cache. The routine **drop_update_class** classifies coalesced update messages and their word update components received for the invalidated block. All other routines and variables in our algorithm for a WU protocol (with or without coalescing write buffers) can be used without modification for this type of hybrid protocol.

```
#define NOTIME -1

void read_miss_class(proc_id, block_id, word)
int proc_id, block_id, word;
{
  if (!Cold[proc_id, block_id]) {
    Cold[proc_id, block_id] = True;
    M_cold ++;
  } else
    if (Dropped[proc_id, block_id]) {
      M_dropped ++;
      Dropped[proc_id, block_id] = False;
    } else M_evict ++;
  read_hit_class(proc_id, block_id, word);
}

void drop_block_class(proc_id, block_id)
int proc_id, block_id;
{
  Dropped[proc_id, block_id] = True;
}

void drop_update_class(proc_id, block_id, nwr)
int proc_id, block_id, nwr;
{
  foreach word in block_id {
    time = msg[proc_id, word].time;
    if (time != NOTIME) {
      foreach wrd in block_id
        if (time == msg[proc_id, wrd].time) {
          class_individual_upd(proc_id, wrd);
          msg[proc_id, wrd].time = NOTIME;
          First[proc_id, wrd] = False;
        }
      class_coalesced_msg(proc_id, word);
    }
  }
  U_drop = U_drop + nwr;
  Msg_drop++;
}
```

Figure A.7: Classification of data and update traffic under a hybrid protocol.

Bibliography

[Agarwal, 1991] A. Agarwal, "Limits on Interconnection Network Performance," *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, Oct 1991.

[Agarwal, 1992] A. Agarwal, "Performance Tradeoffs in Multithreaded Processors," *IEEE Transactions on Parallel and Distributed Systems*, 3(5):525–539, Sept 1992.

[Agarwal *et al.*, 1995] A. Agarwal, R. Bianchini, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The MIT Alewife Machine: Architecture and Performance," In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[Agarwal and Gupta, 1988] A. Agarwal and A. Gupta, "Memory Reference Characteristics of Multiprocessor Applications under Mach," *Performance Evaluation Review*, 16(1):215–225, May 1988, originally published at SIGMETRICS '88.

[Alverson *et al.*, 1990] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System," In *Proceedings of the 1990 International Conference on Supercomputing*, 1990.

[Archibald and Baer, 1986] J. Archibald and J.-L. Baer, "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, 4(4):273–298, Nov 1986.

[Baer and Chen, 1991] J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," In *Proceedings of Supercomputing '91*, pages 176–186, 1991.

[Bailey et al., 1994] D. Bailey et al., "The NAS Parallel Benchmarks," Technical Report RNR-94-007, NASA Ames Research Center, March 1994.

[BBN, 1989] BBN Advanced Computers Inc., *Inside the TC2000*, 1989.

[Bennett *et al.*, 1990] J. K. Bennett, J. B. Carter, and W. Zwaenepoel, "Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence," In *Proceedings of the 2nd ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 168–176, March 1990.

[Bianchini and Brown, 1993] R. Bianchini and C. M. Brown, "Parallel Genetic Algorithms on Distributed-Memory Architectures," In *Transputer: Research and Applications. NATUG-6*, pages 67–82, May 1993.

[Bianchini and LeBlanc, 1992] R. Bianchini and T. J. LeBlanc, "Software Caching on Cache-Coherent Multiprocessors," In *Proceedings of the 4th Symposium on Parallel and Distributed Processing*, Dallas, TX, December 1992.

[Bisiani and Ravishankar, 1990] R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared-Memory System," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.

[Bolosky *et al.*, 1989] W. J. Bolosky, R. P. Fitzgerald, and M. L. Scott, "Simple But Effective Techniques for NUMA Memory Management," In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989.

[Boothe and Ranade, 1993] R. Boothe and A. Ranade, "Performance on a Bandwidth Constrained Network: How much bandwidth do we need," In *Proceedings of Supercomputing '93*, pages 906–915, 1993.

[Callahan *et al.*, 1991] D. Callahan, K. Kennedy, and A. Porterfield, "Software Prefetching," *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[Carter *et al.*, 1991] J. B. Carter, J. K. Bennett, and W. Zwaenepoel, "Implementation and Performance of Munin," In *Proceedings of the 13th Symposium on Operating Systems Principles*, October 1991.

[Chandra *et al.*, 1994] R. Chandra, S. Devine, B. Verghese, A. Gupta, and M. Rosenblum, "Scheduling and Page Migration for Multiprocessor Compute Servers," In *Proceedings of the 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[Cheriton *et al.*, 1991a] D. R. Cheriton, H. A. Goose, and P. D. Boyle, "Paradigm: A Highly Scalable Shared-Memory Multicomputer Architecture," *IEEE Computer*, 24:33–46, February 1991.

[Cheriton *et al.*, 1991b] D. R. Cheriton, H. A. Goosen, and P. Machanick, "Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience," In *Proceedings of the International Symposium on Shared-Memory Multiprocessing*, pages 109–118, Tokyo, Japan, April 1991.

[Cox and Fowler, 1989] A. L. Cox and R. J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM," In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, 1989.

[Dackland *et al.*, 1992] K. Dackland, E. Elmroth, B. Kagstrom, and C. Van Loan, "Parallel Block Matrix Factorizations on the Shared-Memory Multiprocessor IBM 3090 VF/600J," *The International Journal of Supercomputer Applications*, 6(1):69–97, Spring 1992.

[Dahlgren *et al.*, 1993] F. Dahlgren, M. Dubois, and P. Stenstrom, "Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors," In *Proceedings of the 1993 International Conference on Parallel Processing*, August 1993.

[Dahlgren *et al.*, 1994] F. Dahlgren, M. Dubois, and P. Stenstrom, "Combined Performance Gains of Simple Cache Protocol Extensions," In *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994.

[Dahlgren and Stenstrom, 1995] F. Dahlgren and P. Stenstrom, "Effectiveness of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors," In *Proceedings of the 1st IEEE Symposium on High-Performance Computer Architecture*, January 1995.

[Dahlgren and Stenstrom, 1994] F. Dahlgren and P. Stenstrom, "Reducing the Write Traffic for a Hybrid Cache Protocol," In *Proceedings of the 1994 International Conference on Parallel Processing*, August 1994.

[Dally, 1990] W. J. Dally, "Performance Analysis of k-ary n-cube Interconnection Networks," *IEEE Transactions on Computers*, 39:775–785, June 1990.

[Davis *et al.*, 1991] H. Davis, S. R. Goldschmidt, and J. Hennessy, "Multiprocessor Simulation and Tracing Using Tango," In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II–99 – II–107, August 1991.

[Dongarra *et al.*, 1992] J. Dongarra, R. van de Geijn, and D. Walker, "A Look at Scalable Dense Linear Algebra Libraries," In *Proceedings of the 1992 Scalable High Peformance Computing Conference*, pages 372–379, 1992.

[Dubnicki, 1993] C. Dubnicki, *The Effects of Block Size on the Performance of Coherent Caches in Shared-Memory Multiprocessors*, PhD thesis, Department of Computer Science, University of Rochester, July 1993.

[Dubois *et al.*, 1988] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, 21(2):9–21, Feb 1988.

[Dubois *et al.*, 1993] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom, "The Detection and Elimination of Useless Misses in Multiprocessors," In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 88–97, May 1993.

[Eggers and Jeremiassen, 1991] S. J. Eggers and T. E. Jeremiassen, "Eliminating False Sharing," In *Proceedings of the 1991 International Conference on Parallel Processing*, 1991.

[Eggers and Katz, 1989] S. J. Eggers and R. H. Katz, "The Effect of Sharing on the Cache and Bus Performance of Parallel Programs," In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, April 1989.

[Eggers and Katz, 1988] S. J. Eggers and R. H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherency Protocol Evaluation," In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 373–383, May 1988.

[Fu and Patel, 1992] J. W. C. Fu and J. H. Patel, "Stride Directed Prefetching in Scalar Processors," In *Proceedings of the 25th International Symposium on Microarchitecture*, pages 102–110, December 1992.

[Gallivan *et al.*, 1990] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh, "Parallel Algorithms for Dense Linear Algebra Computations," *SIAM Review*, 32(1):54–135, March 1990.

[Gharachorloo *et al.*, 1990] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy, "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors," In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[Goodman, 1983] James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," In *Proceedings of the 10th International Symposium on Computer Architecture*, pages 124–131, 1983.

[Gottlieb *et al.*, 1983] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, C-32(2):175–189, Feb 1983.

[Gupta *et al.*, 1991] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber, "Comparative Evaluation of Latency Reducing and Tolerating Techniques," In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[Gupta and Weber, 1992] A. Gupta and W.-D. Weber, "Cache Invalidation Patterns in Shared-Memory Multiprocessors," *IEEE Transaction on Computers*, 41(7):794–810, July 1992.

[Jouppi, 1993] N. P. Jouppi, "Cache Write Policies and Performance," In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 191–201, May 1993.

[Karlin *et al.*, 1988] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive Snoopy Caching," *Algorithmica*, 3:79–119, 1988.

[Kendall Square Research Corp., 1992] Kendall Square Research Corp., *KSR1 Principles of Operation*, Kendall Square Research Corporation, 170 Tracer Lane, Waltham, MA, 1992.

[Khera *et al.*, 1993] V. Khera, R. P. LaRowe, Jr., and C. S. Ellis, "An Architecture-Independent Analysis of False Sharing," Technical Report CS-1993-13, Department of Computer Science, Duke University, October 1993.

[Lam *et al.*, 1991] M. Lam, E. Rothberg, and M. E. Wolf, "The Cache Performance and Optimizations of Blocked Algorithms," In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[LaRowe Jr. and Ellis, 1991] R. P. LaRowe Jr. and C. S. Ellis, "Experimental Comparison of Memory Management Policies for NUMA Multiprocessors," *ACM Transactions on Computer Systems*, 9(4):319–363, Nov 1991.

[LeBlanc, 1988] T. J. LeBlanc, "Problem Decomposition and Communication Tradeoffs in a Shared-Memory Multiprocessor," In Martin Schultz, editor, *Numerical Algorithms for Modern Parallel Computer Architectures, IMA Volumes in Mathematics and its Applications Volume 13*, pages 145–163. Springer-Verlag, 1988.

[Lee *et al.*, 1987] R. L. Lee, P.-C. Yew, and D. H. Lawrie, "Multiprocessor Cache Design Considerations," In *Proceedings of the 14th Annual Symposium on Computer Architecture*, pages 253–262, June 1987.

[Lenoski *et al.*, 1990] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor," *Proceedings of the 17th International Symposium on Computer Architecture*, pages 148–159, May 1990.

[Lenoski *et al.*, 1993] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The DASH Prototype: Logic Overhead and Performance," *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, Jan 1993.

[Lilja, 1993] D. J. Lilja, "Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons," *ACM Computing Surveys*, 25(3):303–338, September 1993.

[Marchetti *et al.*, 1995] M. Marchetti, L. I. Kontothanassis, R. Bianchini, and M. L. Scott, "Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems," In *To appear in Proceedings of the International Parallel Processing Symposium '95*, April 1995.

[Markatos and LeBlanc, 1994] E. P. Markatos and T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, April 1994.

[McCreight, 1984] E. M. McCreight, "The Dragon Computer System, an Early Overview," In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, July 1984.

[Mowry and Gupta, 1991] T. Mowry and A. Gupta, "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors," *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[Mowry *et al.*, 1992] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–75, October 1992.

[Ni and McKinley, 1993] L. M. Ni and P. K. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, 26(2):62–76, February 1993.

[Ortega and Romine, 1988] J. M. Ortega and C. H. Romine, "The ijk Forms of Factorization Methods II. Parallel Systems," *Parallel Computing*, 7:149–162, 1988.

[Padua and Wolfe, 1986] D. A. Padua and M. J. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, 29(12), 1986.

[Palacharla and Kessler, 1994] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[Papamarcos and Patel, 1984] Mark S. Papamarcos and Janak H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[Patel and Harrison, 1988] N. M. Patel and P. G. Harrison, "On Hot-Spot Contention in Interconnection Networks," *Performance Evaluation Review*, 16(1):114–123, May 1988, Originally published at SIGMETRICS '88.

[Pfister *et al.*, 1985] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss, "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, Aug 1985.

[Pfister and Norton, 1985] G. F. Pfister and V. Alan Norton, "'Hot Spot' Contention and Combining in Multistage Interconnection Networks," *IEEE Transactions on Computers*, C-34(10):943–948, October 1985.

[Przybylski, 1990] S. Przybylski, "The Performance Impact of Block Sizes and Fetch Strategies," In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 160–169, Seattle, WA, 1990.

[Singh *et al.*, 1992] J. P. Singh, W-D. Weber, and A. Gupta, "SPLASH: Stanford Parallel Applications for Shared-Memory," *Computer Architecture News*, 20(1):5–44, March 1992.

[Smith, 1978a] A. J. Smith, "Sequential Program Prefetching in Memory Hierarchies," *IEEE Computer*, 11(12):7–21, December 1978.

[Smith, 1987] A. J. Smith, "Line (Block) Size Choice for CPU Cache Memories," *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[Smith, 1978b] B. J. Smith, "A Pipelined, Shared Resource MIMD Computer," In *Proceedings of the 1978 International Conference on Parallel Processing*, 1978.

[Smith and Weiss, 1994] J. E. Smith and S. Weiss, "PowerPC 601 and Alpha 21064: A Tale of Two RISCs," *IEEE Computer*, 27(6):46–58, June 1994.

[Thacker *et al.*, 1992] Charles P. Thacker, David G. Conroy, and Lawrence C. Stewart, "The Alpha Demonstration Unit: A High-Performance Multiprocessor for Software and Chip Development," *Digital Technical Journal*, 4(4):51–65, 1992.

[Thacker and Stewart, 1987] Charles P. Thacker and Lawrence C. Stewart, "Firefly: a Multiprocessor Workstation," In *Proceedings of the 2nd International Conference on Architectual Support for Programming Languages and Operating Systems*, pages 164–172, Palo Alto, CA, October 1987.

[Torrellas *et al.*, 1994] J. Torrellas, M. S. Lam, and J. L. Hennessy, "False Sharing and Spatial Locality in Multiprocessor Caches," *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[Vaswani and Zahorjan, 1991] R. Vaswani and J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors," In *Proceedings of the 13th Sumposium on Operating Systems Principles*, pages 26–40, October 1991.

[Veenstra and Fowler, 1992] J. E. Veenstra and R. J. Fowler, "A Performance Evaluation of Optimal Hybrid Cache Coherency Protocols," In *Proceedings of the 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, Boston, MA, October 1992.

[Veenstra and Fowler, 1994a] J. E. Veenstra and R. J. Fowler, "MINT: A Front End for Efficient Simulation of Shared-Memory Multiprocessors," In *Proceedings of the 2nd International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, 1994.

[Veenstra and Fowler, 1994b] J. E. Veenstra and R. J. Fowler, "The Prospects for On-Line Hybrid Coherency Protocols on Bus-Based Multiprocessors," Technical Report 490, Department of Computer Science, University of Rochester, March 1994.

[Wilson and LaRowe, 1992] A. W. Wilson and R. P. LaRowe, "Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture," *Journal of Parallel and Distributed Computing*, 15(4):351–367, 1992.

[Wittie and Maples, 1989] L. Wittie and C. Maples, "MERLIN: Massively Parallel Heterogeneous Computing," In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–142 – I–150, August 1989.

[Yew *et al.*, 1987] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie, "Distributing Hot-Spot Addressing in Large-Scale Multiprocessors," *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.